**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

**Факультет інформатики та обчислюваної техніки**

**Кафедра автоматики і управління в технічних системах**

«На правах рукопису»
УДК 004.4

До захисту допущено:
Завідувач кафедри
Олександр РОЛІК
«___»_____20__ р.

# Магістерська дисертація

**на здобуття ступеня магістра**

**за освітньо-науковою програмою «Інженерія програмного забезпечення комп'ютерних систем»**

**зі спеціальності 121 «Інженерія програмного забезпечення»**

**на тему: «Вебзастосунок для онлайн взаємодії тренерів з клієнтами»**

Виконав:
студент VI курсу, групи ІТ-91мн
Олійник Олександр Сергійович _____

Керівник:
Професор кафедри, д.т.н., професор,
Корнієнко Богдан Ярославович

Рецензент:
Доцент кафедри технічних та програмних
засобів автоматизації ІХФ, к.т.н., доцент
Ладієва Леся Ростиславівна

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

**Національний технічний університет України**
**«Київський політехнічний інститут імені Ігоря Сікорського»**
**Факультет інформатики та обчислюваної техніки**
**Кафедра автоматики і управління в технічних системах**

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Олександр РОЛІК
«___»_____20__ р.

**ЗАВДАННЯ**
**на магістерську дисертацію студенту**
**Олійнику Олександру Сергійовичу**

1. Тема дисертації «Вебзастосунок для онлайн взаємодії тренерів з клієнтами», науковий керівник дисертації Корнієнко Богдан Ярославович, к.т.н., професор, затверджені наказом по університету від «12» березня 20 21 р. № 809-с

2. Термін подання студентом дисертації 11 травня 2021 р.

3. Об'єкт дослідження: *процес взаємодії тренерів та спортсменів*

4. Предмет дослідження: *застосування сучасних технологій створення веб-сервісу для розробки веб-застосунку для взаємодії тренерів та спортсменів.*

5. Перелік завдань, які потрібно розробити: *1. Провести аналіз IT-проектів за схожим принципом роботи, та визначити основні переваги та недоліки; 2. Визначити загальний архітектурний підхід та основні компоненти для побудови веб-застосунку; 3. Визначення технології проектування серверної частини.*

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: *зображення графіки, таблиці*

7. Орієнтовний перелік публікацій: *доповіді на конференціях*

8. Консультанти розділів дисертації*

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| | | | |

9. Дата видачі завдання 01 лютого 2021 р.

Календарний план

| № з/п | Назва етапів виконання магістерської дисертації | Термін виконання етапів магістерської дисертації | Примітка |
|---|---|---|---|
| | Порівняльний аналіз існуючих рішень | 02.02.2021-10.02.2021 | |
| | Визначення основних вимог до системи | 11.02.2021-15.02.2021 | |
| | Визначення сценаріїв використання системи | 16.02.2021-22.02.2021 | |
| | Вибір технологій для розробки | 23.02.2021-28.02.2021 | |
| | Реалізація бізнес-логіки | 01.03.2021-28.03.2021 | |
| | Розробка інтерфейсу користувача | 29.03.2021-05.04.2021 | |
| | Аналіз виконаної роботи | 06.04.2021-08.04.2021 | |
| | Оформлення текстового матеріалу | 09.04.2021-19.04.2021 | |
| | Оформлення графічного матеріалу | 20.04.2021-04.05.2021 | |
| | Подача дисертації на перевірку | 05.05.2021-07.05.2021 | |

Студент Олександр ОЛІЙНИК

Науковий керівник Богдан КОРНІЄНКО

# РЕФЕРАТ

Обсяг роботи: 113 сторінок

Кількість ілюстрацій: 43

Кількість таблиць: 15

Кількість джерел за переліком посилань: 34

Актуальність роботи полягає в доцільності використання даної платформи у наш час, у зв’язку з епідеміологічною ситуацією.

Мета: Підвищення ефективності онлайн взаємодії тренерів з клієнтами (скорочення часу на передачу інформації в взаємодії).

Об’єкт дослідження: процес взаємодії тренерів та спортсменів

Предмет дослідження: застосування сучасних технологій створення веб-сервісу для розробки веб-застосунку для взаємодії тренерів та спортсменів.

Методи дослідження – Під час розв’язку поставлених задач було використано методи порівняння, за допомогою яких було проаналізовано існуючі рішення та аналогів.

Наукова новизна – наукова новизна роботи полягає у розробці вебзастосуноку спрямованого на підвищення ефективності взаємодії тренерів з клієнтами через перехід їх взаємодію в онлайн.

Практична цінність – розроблено вебзастосунок з використання сучасних технологічних рішень, що вирішує актуальні проблеми та актуальні недоліки конкурентів.

Ключові слова: клієнт-сервер, тренер, клієнт, база даних, взаємодія.

# ABSTRACT

Volume of work: 113 pages

Number of illustrations: 43

Number of tables: 15

Number of sources on the list of links: 34

The urgency of the work lies in the feasibility of using this platform in our time, due to the epidemiological situation.

Objective: To increase the efficiency of online interaction of trainers with clients (reduction of time for information transfer in interaction).

Object of research: the process of interaction between coaches and athletes

Subject of research: the use of modern technologies for creating a web service for the development of a web application for the interaction of coaches and athletes.

Research methods - In solving the tasks, comparison methods were used, which were used to analyze existing solutions and analogues.

Scientific novelty - scientific novelty of work consists in development of the web application directed on increase of efficiency of interaction of trainers with clients through transition of their interaction to online.

Practical value - developed a web application using modern technological solutions that solves current problems and current shortcomings of competitors.

Keywords: client-server, trainer, client, database, interaction.

ЗМІСТ

# ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

API – application programming interface;

ООП – об'єктно-орієнтоване програмування;

FP – функціональне програмування;

FRP – функціональне реактивне програмування;

ORM – об’єктно-реляційне відображення;

БД – база даних;

СУБД – система управління базами даних;

DOM – модель об’єкта документа;

CBA – Component  Based Architecture;

# ВСТУП

На сьогоднішній день доступно безліч сервісів і додатків для людей, які цікавляться фітнесом. Кожен з цих додатків вирішує певні завдання користувача, і сервіси можна поділити на лічильники калорій і додатки з базами вправ.

Проблема полягає в тому, що для досягнення мети користувачу необхідно мати кілька додатків на своєму телефоні, так як вони не містять необхідного функціоналу.

Актуальність роботи полягає в доцільності використання даної платформи у наш час, у зв’язку з епідеміологічною ситуацією.

Мета: Підвищення ефективності онлайн взаємодії тренерів з клієнтами (скорочення часу на передачу інформації в взаємодії).

Завдання дослідження:

1. Провести аналіз IT-проектів за схожим принципом роботи, та визначити основні переваги та недоліки.

2. Визначити загальний архітектурний підхід та основні компоненти для побудови веб-застосунку.

3. Визначення технології проектування серверної частини.

Об’єкт дослідження: процес взаємодії тренерів та спортсменів

Предмет дослідження: застосування сучасних технологій створення веб-сервісу для розробки веб-застосунку для взаємодії тренерів та спортсменів.

Методи дослідження – Під час розв’язку поставлених задач було використано методи порівняння, за допомогою яких було проаналізовано існуючі рішення та аналогів.

Наукова новизна – наукова новизна роботи полягає у розробці вебзастосуноку спрямованого на підвищення ефективності взаємодії тренерів з клієнтами через перехід їх взаємодію в онлайн.

Практична цінність – розроблено вебзастосунок з використання сучасних технологічних рішень, що вирішує актуальні проблеми та актуальні недоліки конкурентів.

Апробація результатів дослідження: основні положення роботи були висвітлені на наступних наукових конференціях:

1. XVII Міжнародна науково-практична конференція "Європейська наука у 21 столітті - 2021" - 07-15 травня 2021 року, “Nauka I studia”,Przemysl , Польша.

2. XVII міжнародна наукова практична конференція "останні наукові досягнення - 2021" - 17-25 березня 2021: "Біле МІСТО-БГ", Софія

# 1 ОГЛЯД ІСНУЮЧИХ СИСТЕМ

## 1.1 Опис предметної області

Веб застосунок спрямований на створення єдиної системи для взаємодії між тренерами та спортсменами через інтернет. Веб застосунок розрахований на два типи користувачів: «тренер» та «спортсмен», користувач обирає до якого типу він відносить на початку реєстрації в застосунку. Під час реєстрації, в залежності від обраного типу користувача необхідно заповнити поля форми та зареєструватися, далі користувачу буде відправлено лист на електрону адресу, яку було вказано під час реєстрації для підтвердження облікового запису.

Після авторизації користувач потрапляє на сторінку «Моя стрічка», яка містить останні новини або статті інших користувачів, які відстежує користувач. Ці статті можливо переглядати, коментувати, зберігати або вподобати. Також на сторінці є блок з фільтрами, що дозволить швидко знайти інформацію відповідно до потреб користувача. Користувач також має можливість поділитися новиною, або корисною інформацію шляхом написання своєї статті, для цього необхідно перейти на сторінку створення статті в меню. На сторінці розміщена форма з необхідними полями для створення страті.

Основними розділами застосунку є «тренування» та «харчування». В розділі «тренування» розміщенні програми тренувань створенні «тренерами», також є фільтр, що допомагає знати необхідні програми тренувань. Програми тренувань при заходженні на сторінку будуть оптимально підібрані під користувача, відповідно до даних облікового запису. В розділі «харчування» розміщенні раціони харчування, які були розроблені «тренерами». За допомогою фільтру є можливість знайти необхідні раціони харчування. Раціони харчування при заходженні на сторінку будуть підібрані, як і програми тренувань під користувача.

Для створення програм тренувань та раціонів харчування необхідно перейти на сторінку їх створення, що містить конструктори для зручного створення. В конструкторі раціону харчування необхідно створювати страви додаючи різні

продукти, які містяться в базі застосунку, а також можна створювати свої власні страви, якщо таких немає в базі. Можна створювати різні страви і шаблони цілих раціонів, щоб в подальшому використовувати їх для різних клієнтів, змінюючи тільки ввідні дані та параметри. Яка калорійність і яке співвідношення білків, жирів і вуглеводів буде в раціоні вирішує тільки сам користувач. Будь-який раціон можна відправити користувачеві, якщо він зареєстрований на платформі. І якщо не зареєстрований, то ви можете надіслати посилання на раціон, або відправити його на пошту. Аналогічний принцип роботи з конструктором для створення планів тренувань. Використовуючи готову базу вправ, функціонал застосунку дозволяє швидко і просто створювати тренувальні плани і ділитися ними із користувачами. При необхідності можна додати свої власні вправи. Такий підхід до створення раціонів і планів тренувань істотно прискорює весь процес і робить його більш комфортним для сприйняття.

Веб застосунок має також підключену систему оплати, щоб користувачі могли створювати програми тренувань та раціони харчування на платній основі. Таким чином користувачі можуть надавати професійні послуги та отримувати за це кошти.

## 1.2 Аналіз IT-проектів за схожим принципом роботи

При пошуку подібних розробок для рішення поставлених задач було знайдено декілька варіантів систем із схожим принципом роботи.

### 1.2.1 Застосунок «Master Trener»

Застосунок «Master Trener» (рис. 1.2.1) створений для пошуку та підбору персонального тренера відповідно до обраної користувачем дисципліни. Тренер розробляє індивідуальну програму тренувань для занять спортом вдома, або в залі та відповідає на поставленні питання в чаті застосунку [3].

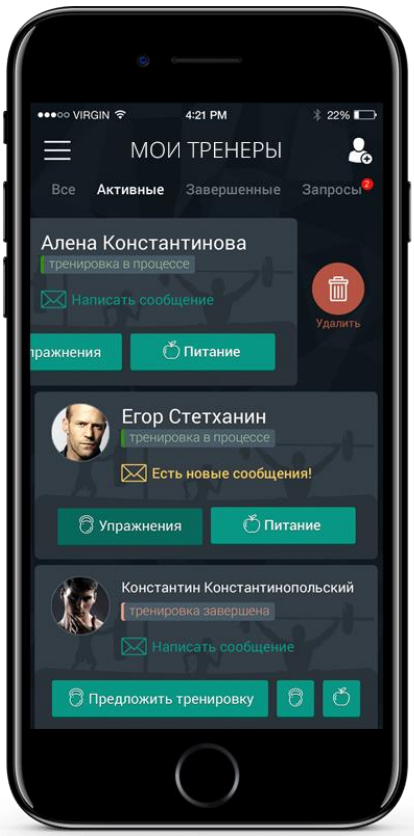



Рисунок. 1.2.1– Застосунок «Master Trener» [3]

До переваг використання описаного застосунку відносяться:

- наявність чату для спілкування з тренером на пряму та в будь-який час;
- анкета користувача;
- пошук тренерів за допомогою системи фільтрів;
- рейтинг тренерів. Дозволяє оцінювати роботи тренера, щоб інші користувачі мали уявлення про його роботу;
- онлайн оплата. Забезпечує зручний та швидкий спосіб розрахунку за послуги;
- інтеграція з соціальними мережами.

До недоліків застосунку відносяться:

- мобільний додаток лише для користувачів IOS;
- відсутній конструктор програм тренувань, користувачам необхідно шукати як виконувати вправи;
- відсутність профільних статей;
- відсутній розділ с раціоном харчування.

### 1.2.2 Онлайн сервіс «Trener.UA»

Trener.UA – безкоштовний онлайн сервіс, за допомогою якого можна знайти тренера як для індивідуальних, так і групових занять в 97 видах спорту по всій Україні (рис. 1.2.2).

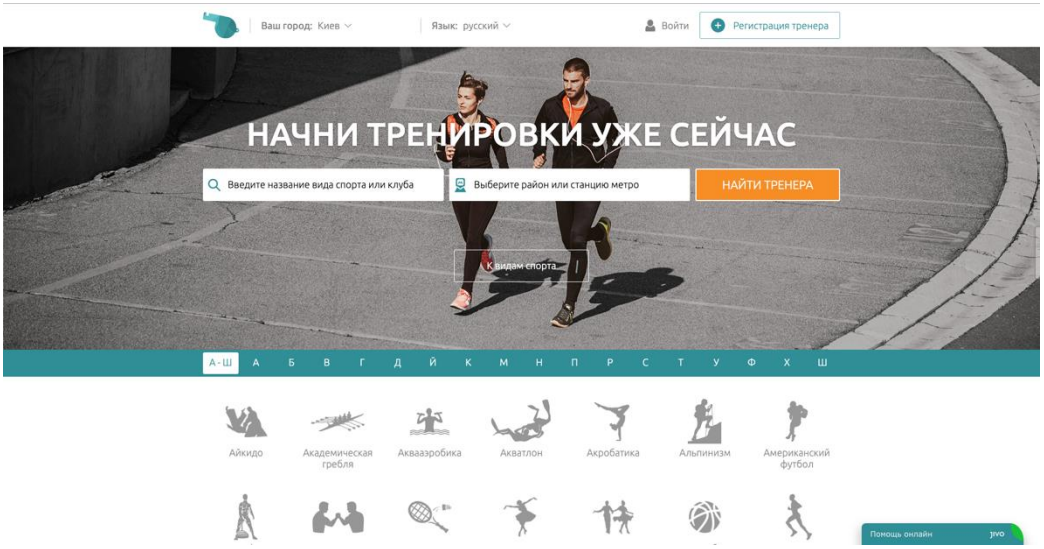



Рисунок 1.2.2 – Онлайн сервіс «Trener.UA» [4]

Сервіс корисний для людей, які ведуть здоровий спосіб життя, і для тренерів, які за допомогою Trener.UA зможуть збільшити кількість клієнтів [4].

Переваги використання:

- сервіс доступний користувачам на всіх платформах;
- велика кількість видів спорту;
- рейтинг тренерів;
- гнучка система фільтрів;
- пошук спортзалів для тренування.

Недоліки:

- відсутній конструктор програм тренувань, користувачам необхідно шукати як виконувати вправи;
- відсутній розділ с раціоном харчування;
- відсутній функціонал онлайн оплати;

- відсутність профільних статей;
- відсутній функціонал онлайн чату.

### 1.2.3 Онлайн фітнес майданчик «FitUnion»

FitUnion – комунікаційний онлайн фітнес майданчик (рис. 1.2.3), на якому будь-який спортсмен знайде професійного тренера для тренувань у фітнес-клубі свого міста або для занять онлайн. Всі прямі контакти тренера або фітнес-клубу розміщені у відкритому доступі на сайті. Є можливість зв'язатися через соціальні мережі, месенджери або зателефонувати безпосередньо.

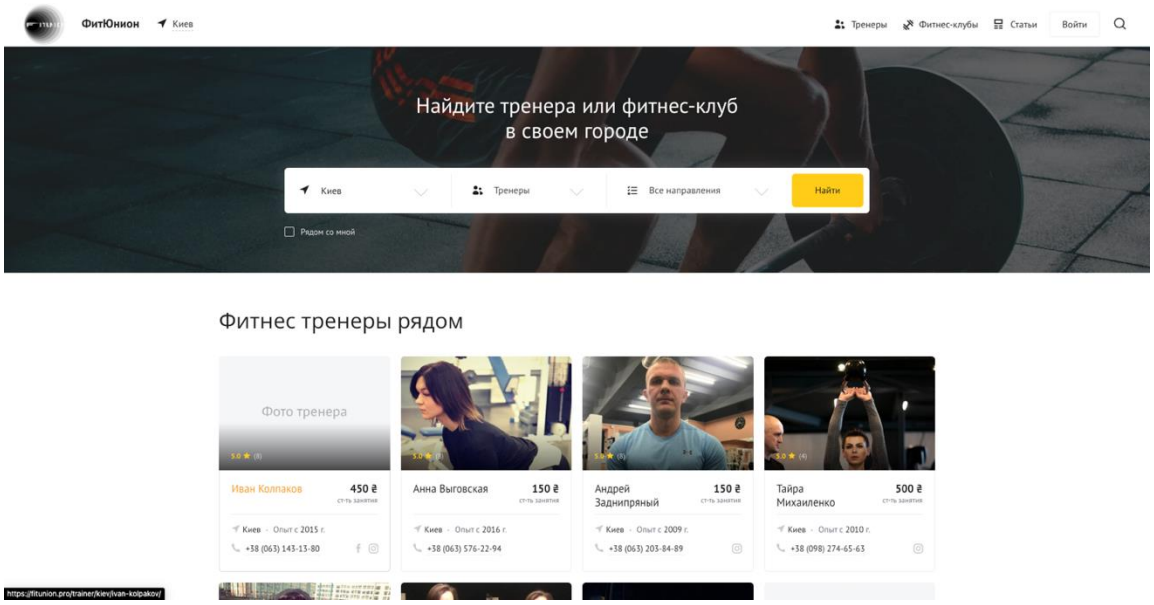



Рисунок 1.2.3 – Онлайн фітнес майданчик «FitUnion» [5]

Відмітка «PRO» говорить про те, що вся інформація про тренера була перевірена адміністратором. Також доступні статті на різні теми фітнесу, що були написанні професійними тренерами. Це додаткова інформація про заняття спортом або правила харчування у відкритому доступі [5].

Переваги використання:

- сервіс доступний користувачам на всіх платформах;
- велика кількість видів спорту;

- рейтинг тренерів;
- гнучка система фільтрів;
- пошук спортзалів для тренування;
- статус PRO для тренерів;
- професійні статті у відкритому доступі;

Недоліки:

- відсутній конструктор програм тренувань, користувачам необхідно шукати як виконувати вправи;
- відсутній розділ с раціоном харчування;
- відсутній функціонал онлайн оплати;
- відсутній функціонал онлайн чату;
- відсутній профіль типу «спортсмен».

## 1.3 Аналіз інформаційного забезпечення

Реалізація рішення проводилося з використанням клієнт-серверної архітектури.

Клієнт-серверна архітектура набула своєї популярності завдяки динамічному розвитку мережі Інтернет та зосередження значної частини інформації в базах даних на серверах.

Клієнт-серверну архітектуру можна означити, як концепцію інформаційної мережі в якій основна частина її ресурсів зосереджена в серверах, обслуговуючих своїх клієнтів.

Дана архітектура визначає такі типи компонентів (рис.1.3.1):

- набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
- набір клієнтів, які використовують сервіси, що надаються серверами;
- мережа, яка забезпечує взаємодію між клієнтами та серверами.

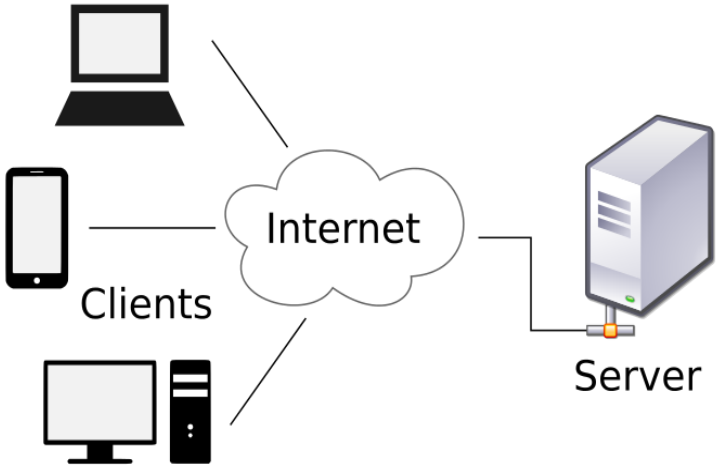



Рисунок 1.3.1 – Схема клієнт-серверної архітектури [1]

Правила взаємодії між клієнтом і сервером називаються протоколом обміну (протоколом взаємодії).

Модель клієнт-серверної взаємодії визначається перш за все розподілом обов'язків між клієнтом та сервером. Логічно можна відокремити три рівні операцій:

- рівень представлення даних, являє собою інтерфейс користувача і відповідає за представлення даних користувачеві і введення від нього керуючих команд;
- прикладний рівень, який реалізує основну логіку застосунку і на якому здійснюється необхідна обробка інформації;
- рівень управління даними, який забезпечує зберігання даних та доступ до них.

При розробці веб застосунку було використано принцип дволанкової клієнт-серверної архітектури. Дволанкова клієнт-серверна архітектура передбачає взаємодію двох програмних модулів — клієнтського та серверного.

## Висновки до розділу 1

Проведено аналіз існуючих рішень, їх функціональні можливості та частини, з яких складаються ці рішення. На основі цього підтверджено визначені вимоги як до системи, так і до окремих її частин.

Визначено загальний архітектурний підхід, в якості якого було вирішено використовувати клієнт-серверну архітектуру, що передбачає взаємодію двох програмних модулів — клієнтського та серверного.

# 2 ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ

Програмний комплекс включає наступні компоненти:

- Nuxt.js – клієнтська частина, призначення для відображення та взаємодії користувачів;
- Nest.js REST API – серверна частина, призначена для обробки та передачі даних до клієнту;
- база данних MYSQL – сховище даних;
- Docker - розгортання та запуск.

## 2.1 Nest.js

Nest (NestJS) – це основа для створення ефективних, масштабованих серверних додатків Node.js. Він використовує прогресивний JavaScript, побудований та повністю підтримує TypeScript і поєднує елементи ООП, FP та FRP. Nest забезпечує рівень абстракції над цими загальними фреймворками Node.js (Express / Fastify), та надає їх API безпосередньо розробнику. Це дає розробникам можливість використовувати безліч сторонніх модулів, доступних для базової платформи.

Nest складається з трьох основних компонентів, але їх можна розширювати: контролер, провайдер, модуль.

Контролери відповідають за обробку вхідних запитів та повернення відповідей клієнту (рис. 2.1.1).

Механізм маршрутизації контролює, який контролер отримує які запити. Часто кожен контролер має більше одного маршруту, і різні маршрути можуть виконувати різні дії [7].

Для того, щоб створити базовий контролер, використовуються класи та декоратори. Декоратори пов'язують класи з необхідними метаданими і дозволяють Nest створювати карту маршрутів (прив'язувати запити до відповідних контролерів).

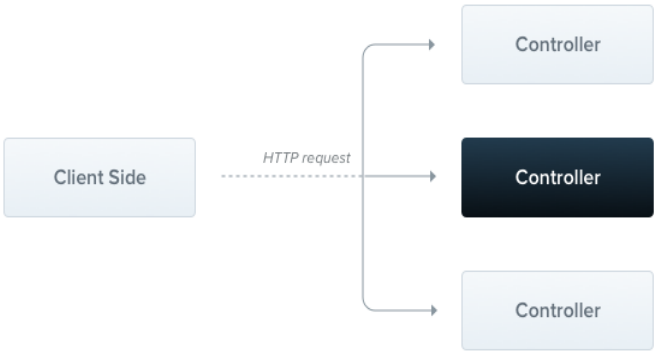



Рисунок 2.1.1 – Схема роботи контролера [7]

Кожна функція всередині контролера може бути анотована наступними деклараторами:

- @Get () – визначення запиту на отримання;
- @Post () – визначення запиту на публікацію ;
- @Delete () – визначення запиту на видалення;
- @Put () – визначення запиту на оновлення.

Провайдери – основне поняття Nest. Основна ідея провайдера полягає в тому, що він може вводити залежності, тобто об'єкти можуть створювати різні взаємозв'язки між собою, а функцію "підключення" екземплярів об'єктів можна в основному делегувати системі виконання Nest (рис. 2.1.2) [8].

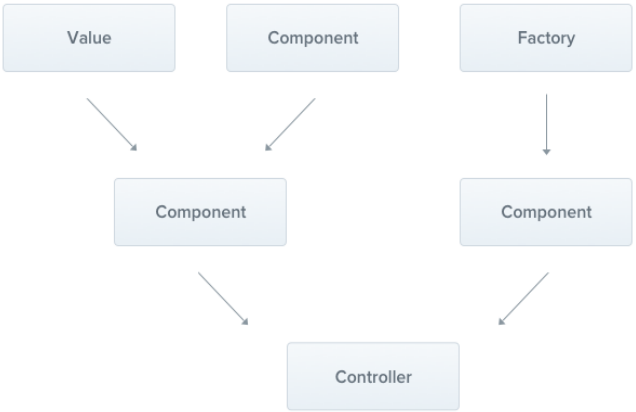



Рисунок 2.1.2 – Схематичне зображення провайдера [8]

Модуль – це клас, котрий коментується декоратором @Module (). Декоратор @Module () надає метадані, які Nest використовує для організації структури програми (рис.2.1.3).

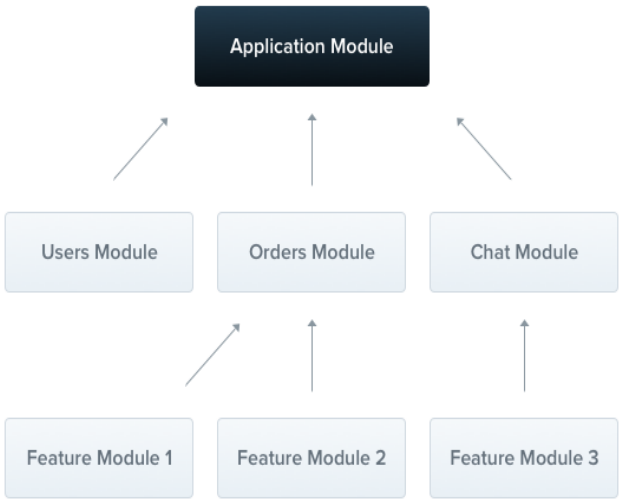



Рисунок 2.1.3 – Схематичне зображення модулей [9]

Кожен додаток має принаймні один модуль, кореневий модуль. Кореневий модуль є відправною точкою, яку Nest використовує для побудови графіку програми – внутрішньої структури даних, яку Nest використовує для вирішення взаємозв’язків та залежностей модуля та постачальника. Отже, використання модулів є ефективним способом упорядкування компонентів. Таким чином, для більшості програм отримана архітектура використовуватиме безліч модулів, кожен з яких містить тісно пов’язаний набір можливостей.

Кожен елемент програми Nest має власний життєвий цикл, який складається з різноманітних подій життєвого циклу, які можна використовувати для забезпечення видимості цих ключових станів та здатності діяти, коли вони виникають.

Події життєвого циклу:

- OnModuleInit – викликається після ініціалізації хост-модуля;
- OnApplicationBootstrap – викликається, коли програма повністю запущена і завантажена;

- OnModuleDestroy – очищення безпосередньо перед тим, як Nest видалить хост-модуль;
- OnApplicationShutdown – закриття програми.

## 2.2 TypeScript

JavaScript був введений як мова для клієнта. Розробка Node.js позначила JavaScript як нову технологію на стороні сервера. Однак, по мірі зростання коду JavaScript, він, як правило, стає неприємнішим, ускладнюючи підтримку та повторне використання коду. Більше того, його нездатність врахувати особливості об'єктної орієнтації, сильна перевірка типу та перевірка помилок під час компіляції заважає JavaScript досягти успіху на рівні підприємства як повноцінна технологія на стороні сервера. Для подолання цієї прогалини було представлено TypeScript.

TypeScript – це строго типізована, об'єктно-орієнтована, компільована мова. Його розробив Андерс Хейлсберг (дизайнер C #) в Microsoft.

Особливості typescript:

1. TypeScript починається з JavaScript і закінчується JavaScript. Весь код TypeScript перетворюється на еквівалент JavaScript для цілей виконання.
2. TypeScript підтримує інші бібліотеки JS. Компільований TypeScript можна використовувати з будь-якого коду JavaScript. Створений TypeScript JavaScript може повторно використовувати всі існуючі фреймворки, інструменти та бібліотеки JavaScript.
3. JavaScript – це TypeScript. Це означає, що будь-який дійсний файл .js може бути перейменований на .ts та скомпільований з іншими файлами TypeScript.
4. TypeScript є портативним. TypeScript є портативним для браузерів, пристроїв та операційних систем. Він може працювати в будь-якому середовищі, на якому працює JavaScript. На відміну від своїх аналогів, TypeScript не потребує виділеної віртуальної машини або певного середовища виконання для виконання.

*TypeScript та ECMAScript*. Специфікація ECMAScript – це стандартизована специфікація мови сценаріїв. Опубліковано шість видань ECMA-262. Версія 6

стандарту має кодову назву "Гармонія". TypeScript узгоджується зі специфікацією ECMAScript6. TypeScript приймає основні мовні особливості зі специфікації ECMAScript5, тобто офіційної специфікації для JavaScript. Мовні функції TypeScript, такі як модулі та орієнтація на основі класу, відповідають специфікації EcmaScript 6. Також TypeScript також охоплює такі функції, як узагальнення та анотації типів, які не є частиною специфікації EcmaScript6(рис. 2.2.1).


Рисунок 2.2.1 – Специфікації TypeScript [14]

До переваг використання TypeScript належать:

1. Компіляція. JavaScript – це інтерпретована мова. Отже, його потрібно запустити, щоб перевірити, чи є він дійсним. Це означає, що ви пишете весь код лише для того, щоб не знайти вихідних даних, на випадок помилки. Отже, вам доведеться витрачати години, намагаючись знайти помилки в коді. Транслятор TypeScript надає функцію перевірки помилок. TypeScript буде компілювати код і генерувати помилки компіляції, якщо виявить певні синтаксичні помилки. Це допомагає виділити помилки перед запуском сценарію.

2. Сильна статична типізація. TypeScript постачається з необов'язковою системою статичного набору тексту та виведення тексту через TLS (мовна служба TypeScript). Тип змінної, оголошений без типу, може визначати TLS на основі її значення.

3. TypeScript підтримує визначення типів для існуючих бібліотек JavaScript. Файл визначення TypeScript (із розширенням .d.ts) забезпечує визначення зовнішніх бібліотек JavaScript. Отже, код TypeScript може містити ці бібліотеки. TypeScript підтримує концепції об'єктно-орієнтованого програмування, такі як класи, інтерфейси, успадкування тощо.

В основі TypeScript є такі три компоненти – мова, компілятор, мовна служба TypeScript(рис. 2.2.2). Мова складається із синтаксису, ключових слів та анотацій типу. Компілятор TypeScript – компілятор TypeScript (tsc) перетворює інструкції, написані на TypeScript, в еквівалент JavaScript.

Мовна служба TypeScript – "Мовна служба" надає додатковий шар навколо конвеєра основного компілятора, що є подібними до редактора програмами. Мовна служба підтримує загальний набір типових операцій редактора, таких як завершення операторів, форматування та окреслення коду тощо.

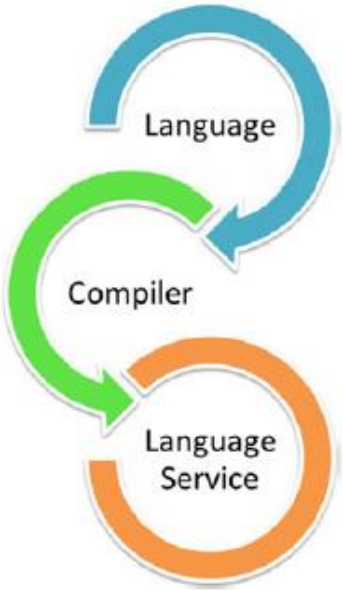



Рисунок 2.2.2 – Схема компонентів TypeScript [14]

Коли компілюється сценарій TypeScript, є можливість створити файл декларації (з розширенням .d.ts), який функціонує як інтерфейс до компонентів скомпільованого JavaScript. Поняття файлів декларацій аналогічно поняттю файлів заголовків, знайдених у C / C ++. Файли оголошень (файли з розширенням .d.ts) забезпечують intellisense для типів, викликів функцій та підтримку змінних для бібліотек JavaScript, таких як jQuery, MooTools тощо.

## 2.3 TypeORM

Фреймворк TypeORM – це об’єктно-реляційне відображення (ORM), в якому частина об'єкта відноситься до домену / моделі у додатку, реляційна частина відноситься до взаємозв'язку між таблицями в системі управління реляційними базами даних (наприклад, Oracle, MySQL, MS-SQL, PostgreSQL тощо) і, частина Картування стосується акту зв’язування моделі та таблиць.

ORM – це тип інструменту, який відображає сутності з таблицями баз даних. ORM забезпечує спрощений процес розробки шляхом автоматизації перетворення об'єктів в таблиці та таблиць в об'єкти.

Оскільки модель слабо пов’язана з іншою частиною програми, її можна змінити без будь-якої жорсткої залежності з іншою частиною програми та легко використовувати її в будь-якому місці програми. TypeORM дуже гнучкий, та абстрагує систему БД подалі від програми і дозволяє нам отримати переваги від використання концепції OOPS.

TypeORM – це об’єктна бібліотека реляційного картографування, що працює в node.js і написана на TypeScript. TypeScript – це вдосконалення JavaScript із необов’язковим набором тексту. TypeScript – це компільована мова. Він не інтерпретується під час виконання. Компілятор TypeScript бере файли TypeScript (.ts) і компілює їх у файли JavaScript (.js).

TypeORM має наступні функції:

- Автоматичнее створення схеми таблиць баз даних на основі моделей;
- Легко вставляти, оновлювати та видаляти об’єкт у базі даних;
- Створення зв’язків (один-до-одного, один-до-багатьох і багато-до-багатьох) між таблицями;
- Надає прості команди CLI.

TypeORM підтримує декілька баз даних, таких як MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, SAP Hana та WebSQL. TypeORM - це проста у використанні ORM для створення нових програм, які підключаються до баз даних. Функціональність TypeORM – це поняття, характерні для RDBMS.

До переваг TypeORM належить: високоякісні та вільно пов'язані додатки; масштабовані програми, легке інтегрувати з іншими модулями, ідеально підходить для будь-якої архітектури від невеликих до корпоративних програм.

## 2.4 MySQL

База даних – це організована колекція структурованої інформації або даних, які зазвичай зберігаються в електронному вигляді в комп’ютерній системі. База даних зазвичай контролюється системою управління базами даних (СУБД). Дані та СУБД разом із пов'язаними з ними програмами називають системою баз даних, часто скороченою до просто бази даних.

Дані в найпоширеніших типах баз даних, що працюють сьогодні, зазвичай моделюються в рядки та стовпці в серії таблиць, щоб зробити обробку та запит даних ефективними. Потім дані можуть бути легко доступні, керовані, модифіковані, оновлені, контрольовані та впорядковані. Більшість баз даних використовують структуровану мову запитів (SQL) для запису та запиту даних.

SQL – це мовна програма, яка використовується в більшості реляційних баз даних для запитів, обробки та визначення даних, а також контролю доступу.

Існує безліч різних типів даних. Вибір найкращої бази даних для конкретної компанії залежить від того, як вона планує використовувати дані.

- Реляційні бази даних. Данні в реляційній базі організовані у таблицях, що містяться в рядках та стопцяї. Реляційна СУБД забезпечує швидкий та ефективний доступ до структурованої інформації.
- Об'єктно-орієнтовані бази даних. Інформація в об'єктно-орієнтованій базі даних представлена у формі об'єкта, як в об'єктно-орієнтованій мові програмування.
- Розподілені бази даних. Складається з двох або більше частин, розташованих на різних серверах. Така база даних може зберігатися на кількох комп'ютерах.

- Сховища даних. Будучи централізованим репозиторієм для даних, зберігання даних представляє собою тип баз даних, спеціально призначений для швидкого виконання запитів та аналізу.

- Бази даних NoSQL. База даних NoSQL (нереляційна база даних) дає змогу зберігати та обробляти структуровані або слабоструктуровані дані (у виведенні з реляційної бази даних, що задає структуру, що містить дані про неї). Популярність бази даних NoSQL зростає за мережею поширення та умовами веб-додатків.

- Графові бази даних. Графічна база даних зберігає дані у контексті сутності та зв'язку між спільнотами.

- Бази даних OLTP. Призначена для виконання бізнес-транзакцій, що виконуються багатьма користувачами.

MySQL – це реляційна система управління базами даних з відкритим вихідним кодом. На даний момент саме ця СУБД одна з найпопулярніших веб-додатків – надання більшості CMS використовує саме MySQL (часто лише без альтернативи), і майже всі веб-фреймворки підтримують MySQL вже на рівні базової конфігурації (без додаткових модулів).

До переваг СУБД MySQL відноситься простота використання, гнучкість, низька вартість використання (відносно платних СУБД), а також масштабність і продуктивність.

MySQL дозволяє зберігати загальнозначущі значення знаків та беззнакових, довжини в 1, 2, 3, 4 та 8 байтів, працює з строковими та текстовими даними фіксованою та перехідною тривалістю, дозволяє здійснювати SQL-команди SELECT, DELETE, INSERT, REPLACE і UPDATE, забезпечує повну підтримку операторів та функцій у SELECT- і WHERE- приватних запитах, працює з GROUP BY та ORDER BY, підтримує групові функції COUNT (), AVG (), STD (), SUM (), MAX () і MIN (), дозволяє використовувати ПРИЄДНАЙТЕСЬ у запрошеннях, у т.ч. LEFT OUTER JOIN та RIGHT OUTER JOIN, підтримує реплікацію, транзакції, роботу із зовнішніми ключами та каскадні зміни на їх основі, а також забезпечує безліч інших функціональних можливостей.

Гнучкість СУБД MySQL забезпечує підтримку великої кількості типових таблиць: користувачі можуть вибрати як таблиці типу MyISAM, підтримку повнотекстового пошуку, так і таблиці InnoDB, підтримку транзакцій на рівні окремих записів. Є та інші типи таблиць, розроблених спільнотою.

СУБД MySQL з'явився в 1995 р. Написана на C і C ++, протестована на багатьох різних компіляторах та працює на різних платформах. С 2010 року розробка та підтримка MySQL здійснює корпорацію Oracle. Продукт розповсюджується як під GNU GPL, так і під власною комерційною ліцензією. Однак за умовами GPL, якщо яка-небудь програма включає вихідні коди MySQL, то і ця програма також повинна бути поширена за ліцензіями GPL. Для небажаних відкрити вихідні тексти своїх програм, як попередньо переглянуто комерційну ліцензію, яка, у доповненні до можливостей розробки під «закритою» ліцензією, забезпечує якісну сервісну підтримку. Повідомлення розробників MySQL створили різні відповіді – Drizzle, OurDelta, Percona Server та MariaDB, всі ці відповіді вже були створені на момент отримання прав на корпорацію MySQL Oracle.

## 2.5 Swagger

Swagger – це набір правил, специфікацій та інструментів для розробки та опису RESTful API з відкритим кодом. Фреймворк Swagger дозволяє розробникам створювати інтерактивну, машинно-зрозумілу документацію API.

Специфікації API зазвичай включають таку інформацію, як підтримувані операції, параметри та результати, вимоги авторизації, доступні кінцеві точки та необхідні ліцензії. Swagger може автоматично генерувати цю інформацію з вихідного коду, просячи API повернути файл документації зі своїх анотацій.

Swagger допомагає користувачам створювати, документувати, тестувати та споживати RESTful веб-сервіси. Його можна використовувати як підхід до розробки API зверху вниз, так і знизу вгору. У методі зверху вниз або проектуванні спочатку Swagger можна використовувати для проектування API перед написанням

будь-якого коду. У методі знизу вгору, або методом, який спочатку кодує, Swagger бере код, написаний для API, і генерує документацію.

## 2.6 SHARP

Типовим варіантом використання цього високошвидкісного модуля Node.js є перетворення великих зображень у загальних форматах у менші, зручні для веб-зображень зображення JPEG, PNG, AVIF та WebP різних розмірів.

Цей модуль підтримує читання зображень JPEG, PNG, WebP, AVIF, TIFF, GIF та SVG. Вихідні зображення можуть бути у форматах JPEG, PNG, WebP, AVIF та TIFF, а також нестиснуті необроблені піксельні дані. Потоки, буферні об'єкти та файлова система можуть використовуватися для введення та виведення. Один вхідний потік можна розділити на кілька конвеєрів обробки та вихідних потоків.

Зміна розміру зображення, як правило, в 4–5 разів швидша, ніж використання найшвидших налаштувань ImageMagick та GraphicsMagick завдяки використанню libvips.

Колірні простори, вбудовані профілі ICC та альфа-канали прозорості обробляються правильно. Передискретизація Ланцоша гарантує, що якість не жертвується швидкістю. Окрім зміни розміру зображення, доступні такі операції, як обертання, вилучення, композитування та гамма-корекція.

## Висновки до розділу 2

Проаналізовано та описано основні модулі для розробки серверної частини вебзастосунку. Основним фреймворком використовується Nestjs для побудови rest api, та додаткові пакети TypeORM, Swagger, Sharp і тд.

# 3 ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ КЛІЄНТСЬКОЇ ЧАСТИНИ

## 3.1 Vue.js

Vue.js – це прогресивний фреймворк для JavaScript, який використовується для побудови веб-інтерфейсів та односторінкових додатків. Vue.js також використовується як для розробки настільних, і мобільних додатків за допомогою платформи Electron. Розширення HTML та база JS швидко зробили Vue улюбленим інтерфейсним інструментом, про що свідчать використання його в Adobe, Behance, Alibaba, Gitlab та Xiaomi.

Модель об’єкта документа (DOM) – це представлення HTML-сторінок з їх стилями, елементами та вмістом сторінки як об’єктами. Об'єкти, що зберігаються як деревоподібна структура, створюються браузером під час завантаження сторінки. Коли користувач взаємодіє зі сторінкою, об’єкти змінюють свій стан, так що браузер повинен оновити інформацію та відобразити її на екрані. Для прискорення швидкості завантаження сторінок та покращення продуктивності роботи програми Vue.js використовує віртуальний DOM.

Ефективність є одним із ключових факторів, який може зумовити вибір фреймворку. Наприклад, під час тестування компонентів DOM, пов’язаних із оновленими даними, Vue.js здається ефективнішим, ніж Angular та React. Звичайно, це далеко не лідируюча позиція, де Vanilla.js вкладений, і еталон включає старі версії фреймворків, тому ви можете це врахувати. Але загальна картина здається оптимістичною.

До переваг використання Vue.js належить:

1. Реактивна двостороння прив'язка даних. Двостороння прив'язка даних – це зв'язок між оновленнями даних моделі та поданням (UI). Зв’язані компоненти містять дані, які можна час від часу оновлювати. За допомогою двостороннього прив’язки даних простіше оновлювати пов’язані компоненти та відстежувати оновлення даних.

2. Маленький розмір. Завантажений zip-файл із фреймворком важить 18 КБ, що забезпечує не тільки швидке завантаження та встановлення бібліотеки, а також позитивно впливає на SEO та UX.

3. Віртуальний рендеринг і продуктивність DOM.

4. Однофайлові компоненти та зручність читання. Кожна частина програми, або веб-сторінки у Vue є компонентом. Компоненти являють собою інкапсульовані елементи інтерфейсу. У Vue.js компоненти можна писати у форматі HTML, CSS та JavaScript, не розділяючи їх на окремі файли.

Розбиття коду програми насправді є архітектурним підходом, який називається Component Based Architecture (CBA), і він також використовується в Angular та React. Такий архітектурний підхід має масу переваг:

- Повторне використання компонентів. Інкапсульовані компоненти – це в основному фрагменти коду, які можна використовувати повторно як шаблони для подібних системних елементів.
- Читабельність коду. Оскільки всі компоненти зберігаються в окремих файлах (а кожен компонент - це лише один файл), код легше читати та розуміти, що полегшує обслуговування та виправлення.
- Добре для модульного тестування. Блок-тестування – це перевірка якості, спрямована на перевірку того, як найменші частини програми працюють самі по собі. Наявність компонентів значно спрощує це завдання.

Vuex – це  шаблон управління станом та бібліотека для програм Vue.js. Він служить централізованим сховищем усіх компонентів програми, з правилами, що гарантують, що стан може мутувати лише передбачувано.

Vue Router – офіційний маршрутизатор для Vue.js. Він глибоко інтегрується з ядром Vue.js, щоб зробити програми для створення односторінкових сторінок за допомогою Vue.js легкими. Особливості включають:

- вкладене відображення маршруту / перегляду;
- модульна конфігурація маршрутизатора на основі компонентів;
- параметри маршруту, запит, символи підстановки;
- ефекти переходу, створені завдяки системі переходів Vue.js;

- зручне управління навігацією;
- посилання з автоматичними активними класами CSS;
- режим історії HTML5 або хеш-режим, з автоматичним відновленням у IE9;
- налаштовувана поведінка прокрутки.

## 3.2 Nuxt

Фреймворки / бібліотеки JavaScript, такі як Vue, можуть запропонувати чудовий досвід користувача під час перегляду сайту. Більшість пропонує спосіб динамічної зміни вмісту сторінки без необхідності щоразу надсилати запит на сервер.

Однак при початковому завантаженні веб-сайту браузер не отримує цілу сторінку для відображення. Натомість йому надсилається купа фрагментів для побудови сторінки (HTML, CSS, інші файли) та інструкції щодо того, як об’єднати їх усі (фреймворк / бібліотека JavaScript), тому потрібна помірна кількість часу, щоб зібрати всю цю інформацію разом перш ніж у вашому браузері є що відобразити. Для того, щоб уникнути такої проблеми необхідно мати на сервері версію платформи / бібліотеки, яка може створити готову до відображення сторінку. Потім надіслати цю повну сторінку браузеру разом із можливістю вносити подальші зміни та все одно мати динамічний вміст сторінки (фреймворк / бібліотека).

Nuxt.js базується на реалізації SSR для популярної бібліотеки React під назвою Next. Переконавшись у перевагах цієї конструкції, для Vue було розроблено подібну реалізацію під назвою Nuxt. Ті, хто знайомий з комбінацією React + Next, виявлять купу подібностей у дизайні та компонуванні програми. Однак Nuxt пропонує спеціальні функції Vue для створення потужного, але гнучкого рішення SSR для Vue.

Nuxt було оновлено до готової до версії 1.0 версії в січні 2018 року та є частиною активної та добре підтримуваної спільноти. Одна з чудових речей полягає в тому, що побудова проекту за допомогою Nuxt не так сильно відрізняється від побудови будь-якого іншого проекту Vue. Насправді він надає купу функцій, які дозволяють створювати добре структуровані бази кодів за скорочений час.

Ще одна важлива річ, на яку слід звернути увагу, - це те, що Nuxt не повинен використовуватися лише для SSR. Він просувається як основа для створення універсальних програм Vue.js і включає команду (nuxt generate) для створення статичних програм Vue, що генеруються, з використанням тієї ж кодової бази. Тож якщо ви боїтесь занурення глибоко в PCP, не панікуйте. Замість цього ви завжди можете створити статичний сайт, користуючись при цьому перевагами функцій Nuxt.

### 3.3. AXIOS

Axios – HTTP-клієнт на основі промісів, який працює як у браузері, так і в середовищі Node.js. Надає єдиний API для роботи з XMLHttpRequests та http-інтерфейсом вузла та обгортає запити, використовуючи поліфіл для синтаксису обіцянок ES6 new. Майже будь-який динамічний проект, який розробляється, в якийсь момент повинен взаємодіяти з RESTFUL API, і використання Axios – це найпростіший спосіб.

При створенні додатку, де потрібно отримувати дані з API, Axios – це простий спосіб це зробити. Також можна використовувати Axios для запитів POST, для зберігання програмою введених даних користувача/даних на сервері. Отже, майже будь-який динамічний веб-сайт, який відображає дані з різних джерел, потребує певного способу для надсилання HTTP-запитів, і Axios є одним з найпопулярніших способів зробити це.

Деякі інші альтернативи Axios включають метод fetch () у Javascript або jQuery AJAX. При отриманні існує два кроки для отримання даних JSON, перший – це здійснення виклику, а другий – виклик методу .json () у цій відповіді .

## 3.4 Nuxt auth

Nuxt-auth – модуль для Nuxt, який спрощує процес та контроль авторизації користувачів в додатку. Модуль дозволяє налаштувати процес автентифікації за допомогою параметра Schemas. Schemas визначають логіку автентифікації. Стратегія – це налаштований примірник схеми. Ви можете мати кілька схем і стратегій у своєму проекті. Модуль містить декілька готових схем автентифікації, також є можливість створити необхідну схему автентифікації.

Наявні схеми автентифікації:

- local – типова схемою на основі облікових даних / маркера для потоків, таких як JWT;
- oauth2 – підтримує різні потоки входу oauth2. Є багато заздалегідь налаштованих постачальників, таких як auth0, які можна використовувати замість того, щоб безпосередньо використовувати цю схему;
- refresh – розширена версія локальної схеми, створена для систем, що використовують оновлення маркера.

## 3.5 BootstrapVue

Бібліотека компонентів інтерфейсу – це (як правило) надійний набір готових компонентів інтерфейсу, таких як кнопки, входи, діалоги тощо. Вони служать будівельними блоками для макетів. Завдяки їх модульній природі ми можемо компонувати компоненти різними способами для досягнення унікальних ефектів. Кожна бібліотека має неповторний зовнішній вигляд, але більшість із них пропонують тематику, а їх компоненти певною мірою налаштовуються.

Переваги бібліотеки компонентів інтерфейсу користувача:

1. Швидкість – включення бібліотеки компонентів інтерфейсу може мати великий вплив на швидкість розробки. Замість того, щоб створювати кожен елемент з нуля, ми просто налаштовуємо та використовуємо існуючі компоненти.

Навіть якщо потрібні додаткові налаштування, це, як правило, набагато швидше, ніж написання власних стилів;

2. Простота використання – компоненти призначені для простоти використання. Створені бібліотеки добре організовані та мають хорошу документацію, за якою легко стежити. Дуже часто для того, щоб це працювало, достатньо лише скопіювати та вставити фрагменти коду;

3. Привабливий вигляд – компоненти виглядають чудово, не вимагаючи додаткових зусиль від розробника;

4. Сумісність – забезпечення сумісності між браузерами та між пристроями є однією з найбільших проблем розвитку інтерфейсу;

5. Доступність – якісні бібліотеки дотримуються вказівок щодо доступності, тому розробникам навіть не потрібно про це думати.

## 3.6 i18n

Інтернаціоналізація (іноді скорочується до «i18n») – це процес планування та впровадження продуктів та послуг, для легкої адаптації до конкретних місцевих мов та культур. Наприклад при створенні продукту для користувачів у США та Франції, без інтернаціоналізації, це означало б створення двох окремих додатків у різних регіонах для роботи в різних доменах (англійська версія на amazingproduct.us та французька версія на amazingproduct.fr).

За допомогою інтернаціоналізації користувачам з обох країн легко використовувати програму у вибраній місцевості (англійською чи французькою). Інтернаціоналізація вашої програми має такі переваги, як:

- Єдиний вихідний код для всіх мов продукту;
- Більше прийняття та задоволення споживачів у країні;
- Це робить обслуговування застосунку простішим;
- Скорочення часу, витрат та зусиль на локалізацію (L10n) ;

Інтернаціоналізація може бути реалізована у Nuxt за допомогою плагіна nuxy-18n. Він легко інтегрує функції локалізації у вашу програму nuxt.js.

## Висновки до розділу 3

Визначено та описано основні компоненти для побудови клієнтської частини веб-застосунку. Основними фреймворком для побудови клієнтської частини є nuxt, з використанням додаткових модулів: vue, vuex, axios, nuxt-auth, i18n, bootstrapVue.

# 4 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ

## 4.1 Користувач

Для взаємодії з користувачами, а саме: створення облікового запису, редагування, видалення, авторизації в застосунку було створено контролер UserController. Він відповідає за обробку всіх запитів які будуть надходити до серверу за адресою /api/user, обробка запитів відбувається в класі UserService (рис 4.1.1).

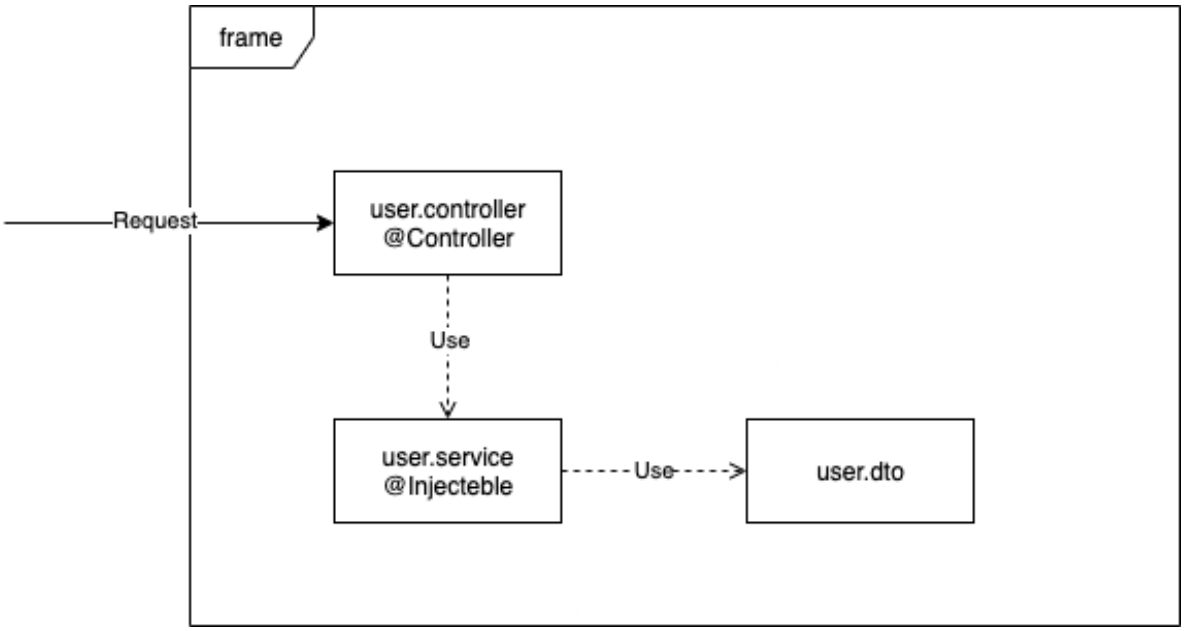



Рисунок 4.1.1 – Схема роботи модуля користувачів

Основні методи контролеру відображено в таблиці 4.1.1

Таблиця 4.1.1 – Методи контролеру

| Методи | Опис |
|---|---|
| findMe | Пошук користувача за email. Виконується при отриманні GET запросу за адресою /api/user, повертає виклик методу userService.findByEmail |

| | |
|---|---|
| update | Оновлення данних користувача. Виконується при отриманні PUT запросу за адресою /api/user, повертає виклик методу userService.update |
| create | Реєстрація користувача в застосунку. Виконується при отриманні POST запросу за адресою /api/users, повертає виклик методу userService.create |
| delete | Видалення користувача з застосунку. Виконується при отриманні DELETE запросу за адресою /api/users/:slug, повертає виклик методу userService.delete |
| login | Авторизація користувача в застосунку. Виконується при отриманні POST запросу за адресою /api/users/login, повертає виклик методу userService.login |

Таблиця 4.1.2 Методи класу UserController

| Методи | Опис |
|---|---|
| findAll | Виконує отримання всіх записів користувачів з бази данних |
| findOne | Виконує отримання одного запису користувача з база даних |
| create | Виконує дадання користувача до бази даних, якщо в базі не має користувача з таким email та інші данні є валідними |
| delete | Виконує видалленя облікового запису користувача з бази даних |
| login | Авторизація користувача в застосунку. Виконується при отриманні POST запросу за адресою /api/users/login, повертає виклик методу userService.login |

| | |
|---|---|
| findById | Виконує пошук корисувача по id в базі даних |
| findByEmail | Виконує пошук корисувача по email в базі даних |
| generateJWT | Виконує створення JWT(Json Web Token) для доступу користувача до застосунку |
| buildUserRO | Виконує створення об'єкту користувача |
| update | Виконує оновлення даних користувача в базі данних, якщо користувач є в базі та данні валідні |

Таблиця 4.1.3 Методи класу UserService

| Поле | Опис |
|---|---|
| id | @PrimaryGeneratedColumn()<br>тип int<br>Ідентифікатор користувача |
| username | @Column()<br>тип string<br>ім'я користувача в застосунку |
| email | @Column()<br>@IsEmail()<br>тип string<br>email користувача |
| firstname | @Column({default: "})<br>тип string<br>имя користувача |
| lastname | @Column({default: "})<br>тип string<br>прізвище користувача |
| bio | @Column({default: "})<br>тип string<br>дані користувача |

| | |
|---|---|
| image | @Column({default: "})<br>тип string<br>картинка профілю користувача |
| password | @Column()<br>тип string<br>пароль користувача для доступу до застосунку |
| articles | @OneToMany(type => ArticleEntity, article => article.author)<br>тип ArticleEntity<br>Публікації користувача |
| exercises | @OneToMany(type => ExerciseEntity, exercise => exercise.author) тип ExerciseEntity[];<br>Вправи користувача |
| diets | @OneToMany(type => DietEntity, diet => diet.author)<br>тип DietEntity[];<br>Дієти користувача |
| favorite | @ManyToMany(type => ArticleEntity)<br>@JoinTable()<br>Тип ArticleEntity[]<br>Публікації які користувач вподобав |

### 4.1.1 Реєстрація та авторизація користувачів

API повинен визначити, що користувачі можуть і не можуть отримати доступ з кожним запитом. Цей процес відомий як авторизація. Щоб знати, що може зробити користувач, спочатку потрібно знати, хто такий користувач. Цей процес відомий як аутентифікація. Наприклад, користувач, ідентифікований як адміністратор, мав би доступ до кінцевих точок, які записують дані, тоді як користувач, ідентифікований як клієнт, не мав.

Веб-програми зазвичай виконують аутентифікацію, просячи користувачів надати набір облікових даних для входу через клієнтську програму, в даному випадку Nuxt. Клієнтська програма надсилає облікові дані на сервер автентифікації, який перевіряє їх. Після успіху сервер автентифікації може обмінюватися інформацією про особу та доступ користувача за допомогою веб-програми. Після успішної авторизації користувачу надається JWT(Json Web Token), коли користувачеві потрібно зробити запит до захищеної кінцевої точки API, клієнтська програма повинна надіслати токен доступу із запитом на API, щоб потім здійснити процес авторизації.

Json Web Token (JWT) – це відкритий стандарт (RFC 7519), який визначає компактний та автономний спосіб безпечної передачі інформації між сторонами як об'єкт JSON. Цю інформацію можна перевірити та довіряти їй, оскільки вона має цифровий підпис. JWT можна підписати, використовуючи секрет (з алгоритмом HMAC) або пару відкритих / приватних ключів, використовуючи RSA або ECDSA. Незважаючи на те, що JWT можуть бути зашифровані, щоб забезпечити таємницю між сторонами, ми зупинимось на підписаних жетонах. Підписані жетони можуть перевірити цілісність вимог, що містяться в ньому, тоді як зашифровані маркери приховують ці вимоги від інших сторін. Коли жетони підписуються за допомогою пар відкритого / приватного ключів, підпис також засвідчує, що лише сторона, що тримає закритий ключ, та, яка підписала його.

Реєстрація користувачів в застосунку відбувається шляхом отримання від

клієнтської частини даних користувача: e-mail та паролю за адресою /api/users. Перед початком реєстрації користувача необхідно провалідувати e-mail, на який хоче здійснити реєстрацію користувач так, як e-mail повинен бути унікальним у всій системі. Після валідації, якщо e-mail є вільним – проводиться реєстрація користувача в системі, інакше відправляється помилка на клієнтську частину. Під час реєстрації користувачів необхідно зберігати в базі пароль в хешованому вигляді, для забезпечення безпеки даних користувачів, для цього використовується bcrypt.

Bcrypt – це алгоритм хешування, який можна масштабувати за допомогою апаратного забезпечення (через конфігуровану кількість раундів). Його повільність і багаторазовість раундів гарантує, що зловмисник повинен задіяти величезні кошти та обладнання, щоб мати можливість зламати паролі. Додайте до цього солі для пароля, і ви зможете бути впевнені, що атака практично неможлива без будь-якої смішної суми коштів або обладнання. bcrypt використовує алгоритм Eksblowfish для хешування паролів. Хоча фаза шифрування Eksblowfish і Blowfish абсолютно однакова, фаза розкладу ключів Eksblowfish гарантує, що будь-який наступний стан залежить як від солі, так і від ключа (пароля користувача), і жоден стан не може бути попередньо обчислений без відома обох. Через цю ключову різницю bcrypt є одностороннім алгоритмом хешування. Ви не можете отримати пароль простого тексту, не знаючи солі, раундів та ключа (пароля). Після успішної реєстрації повертаємо об’єкт користувача

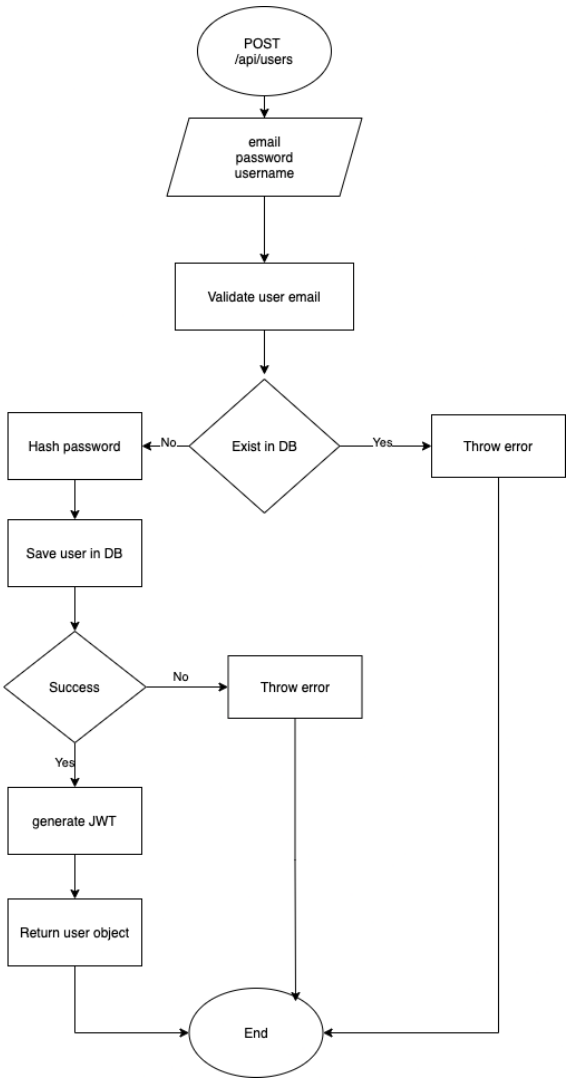



Рисунок 4.1.2 – Схема створення користувача

Авторизація користувачів в застосунку відбувається шляхом отримання від клієнтської частини e-mail та паролю користувача за адресою /api/users/login, які було вказано при реєстрації. Спочатку за email адресою відбувається пошук користувача в базі даних, якщо такий користувач є порівнюємо захешований пароль в базі з паролем, який прийшов від користувача. Якщо паролі співпадають – повертається на клієнтську частину об’єкт користувача (рис. 4.1.1), якщо паролі не співпадають - повертається помилка.

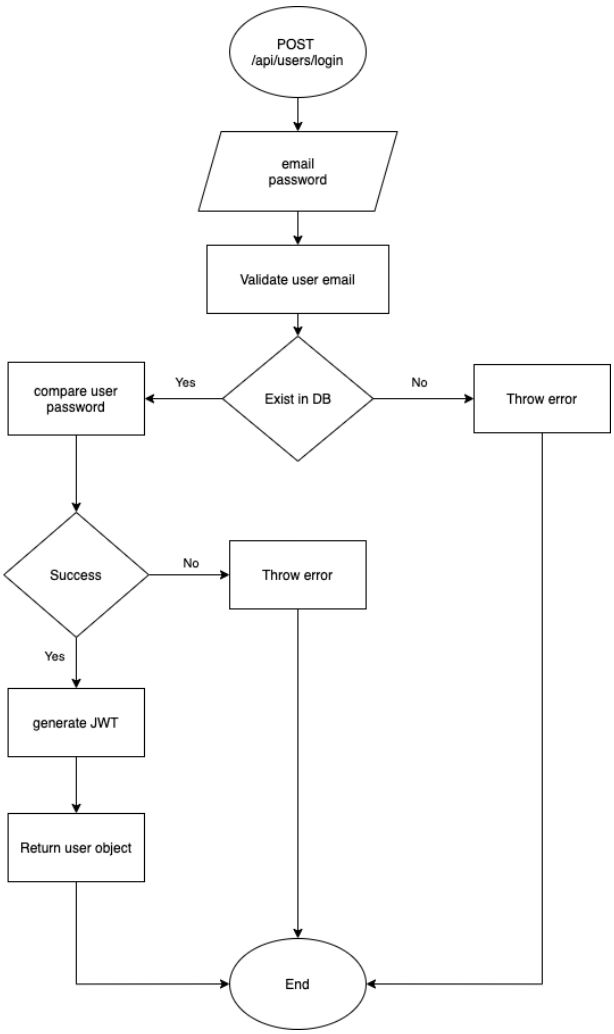



Рисунок 4.1.3 – Схема авторизації користувача

### 4.1.2 Редагування та видалення профілю користувача

Для оновлення даних профілю користувача необхідно відправити запит типу PUT за адресою /api/user. Запит повинен мати заголовок з jwt токеном користувача, для доступу до данного методу, а токож тіло запиту формату json з наступними полями: userId, username, firstname, lastname, email, bio, image. Якщо всі вимоги виконані профіль користувача буде оновлено, та повернуто json об’єкт з оновленими даними користувача (рис.4.1.4).

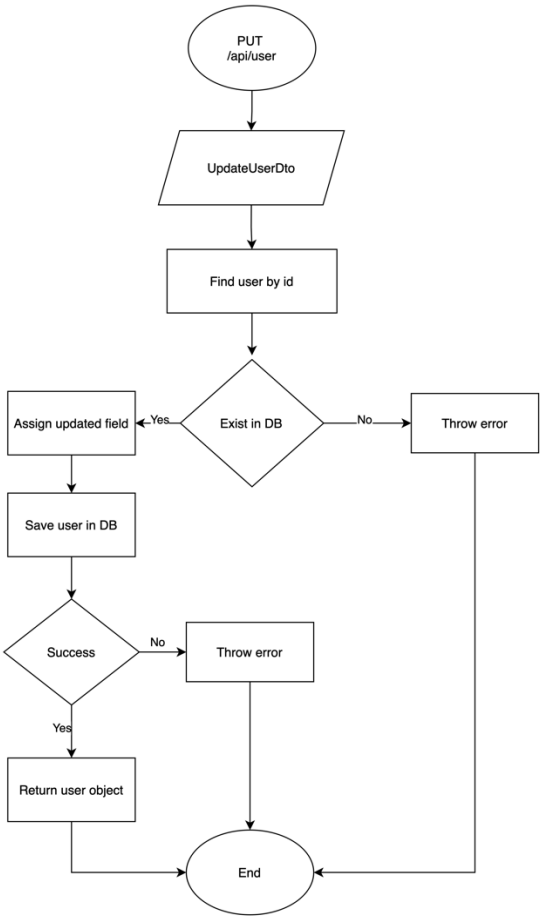



Рисунок 4.1.4 – Схема редагування профілю

Процес видалення користувача схожий з оновленням профілю, проте має відмінності: тип запиту повинен буди DELETE, а сам запит відправлений за адресою /api/users/:slug, де :slug – email користувача. Якщо процес видалення буде виконано, буде повернуто повідомленя про успіх (рис.4.1.5).

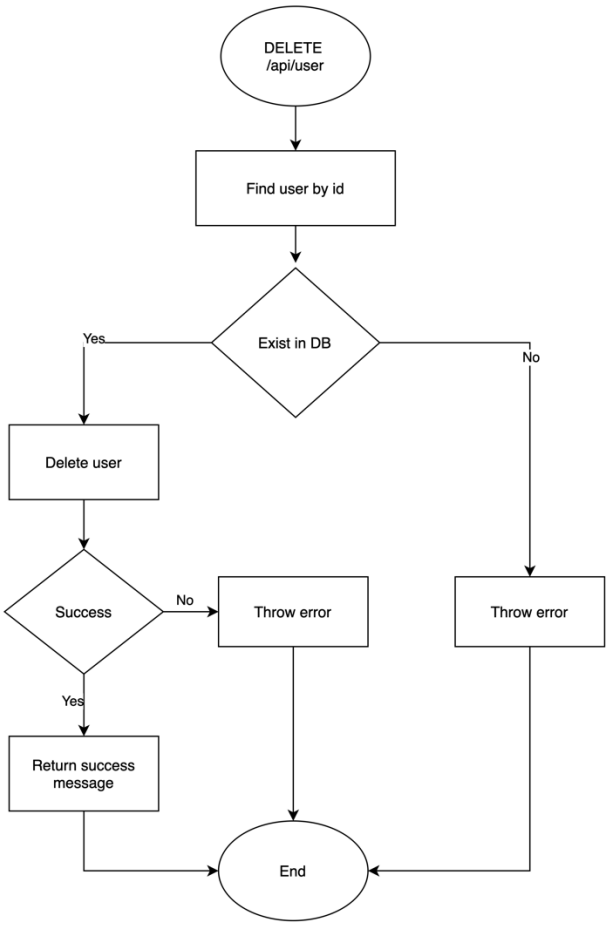



Рисунок 4.1.5 – Схема видалення профілю

## 4.2 Статті

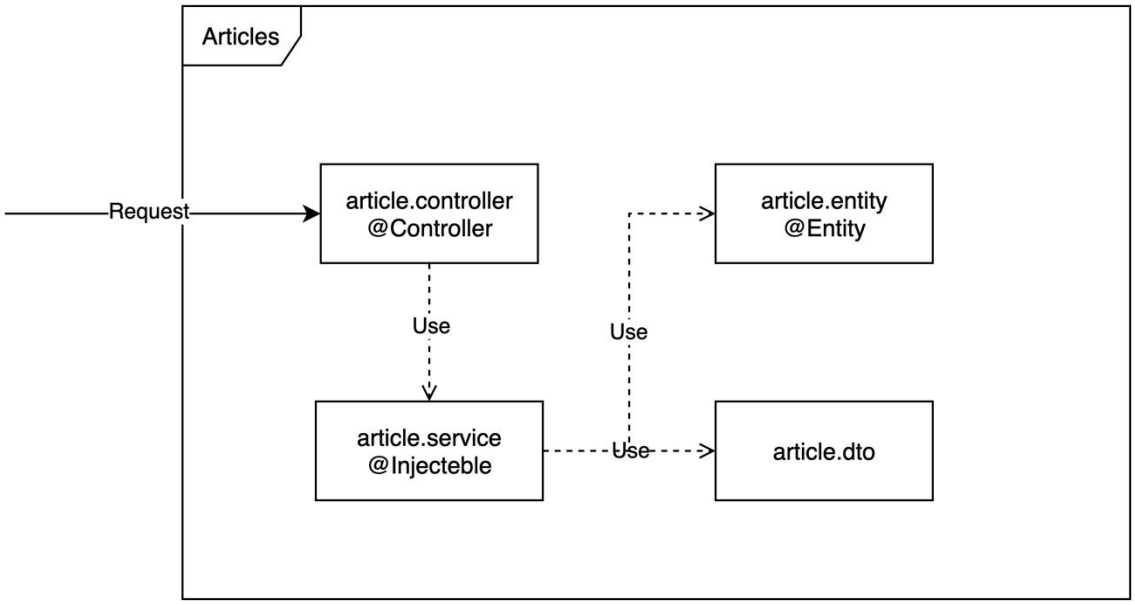



Рисунок 4.2.1 – Схема роботи модуля статей

Таблиця 4.2.1 – Методи класу ArticleController

| Методи | Опис |
|---|---|
| findAll | Отримання всіх статей. Виконується при отриманні GET запросу за адресою /api/articles, повертає виклик методу ArticleService.findAll |
| findOne | Отримання статті по параметру slug. Виконується при отриманні GET запросу за адресою /api/articles/:slug, повертає виклик методу ArticleService.findAll |
| findComments | Отримання коментарів до статті. Виконується при отриманні GET запросу за адресою /api/articles/:slug/comments, повертає виклик методу ArticleService.findComments |
| findExercises | Отримання вправ до статті. Виконується при отриманні GET запросу за адресою |

| | |
|---|---|
| | /api/articles/:slug/exercise, повертає виклик методу ArticleService.findExercises |
| create | Створення статті. Виконується при отриманні POST запросу за адресою /api/articles повертає виклик методу ArticleService.create |
| update | Оновлення статті. Виконується при отриманні PUT запросу за адресою /api/articles/:slug повертає виклик методу ArticleService.update |
| delete | Видалення статті. Виконується при отриманні DELETE запросу за адресою /api/articles:slug повертає виклик методу ArticleService.delete |
| createComment | Створення та додання коментару до статті. Виконується при отриманні POST запросу за адресою /api/articles/:slug/comments повертає виклик методу ArticleService.addComment |
| deleteComment | Видалення коментару. Виконується при отриманні DELETE запросу за адресою /api/articles/:slug/comments/:id повертає виклик методу ArticleService.deleteComment |
| favorite | Додати публікаційю до вродобаних. Виконується при отриманні POST запросу за адресою /api/articles/:slug/favorite повертає виклик методу ArticleService.favorite |
| unFavorite | Видалити публікаційю зі вродобаних. Виконується при отриманні DELETE запросу за адресою /api/articles/:slug/favorite повертає виклик методу ArticleService.unFavorite |
| getFeed | Отримання статей тільки від пидписаних користувачів. Виконується при отриманні GET |

| | |
|---|---|
| | запросу за адресою /api/articles/feed повертає виклик методу ArticleService.findFeed |
| addExercise | Додати вправу до статті. Виконується при отриманні POST запросу за адресою /api/articles/:slug/exercises повертає виклик методу ArticleService.addExercise |
| deleteExercise | Видилити вправу зі статті. Виконується при отриманні DELETE запросу за адресою /api/articles/:slug/exercises/:id повертає виклик методу ArticleService.deleteExercise |

Таблиця 4.2.2 – Методи класу ArticleService

| Методи | Опис |
|---|---|
| findAll | Отримання списку статей з бази даних, які відповідають параметрам: теги, автор, вподобання, кількість та сторінка. |
| findFeed | Отримання списку статей з бази даних, на автори яких користувач підписаний. |
| findOne | Отримання статті з бази даних за слагом. |
| create | Додання статті до бази даних. |
| update | Оновлення статті в базі даних. |
| delete | Видалення статті з бази данних. |
| slugify | Створення слагу статті з назви та чисел. |
| addComment | Додання коментару до статті. |
| deleteComment | Видалення коментару. |
| favorite | Вподобати статтю користувачем та додати до вподобаних. |
| unFavorite | Видалити статтю зі вподобаних. |
| findComments | Отримати всі коментарі до статті. |

| | |
|---|---|
| addExercise | Додання вправи до заданної статті. |
| deleteExercise | Видалення вправи зі статті. |

Таблиця 4.2.3 – Поля класу ArticleEntity

| Поле | Опис |
|---|---|
| id | @PrimaryGeneratedColumn()<br>тип number<br>Ідентифікатор статті |
| slug | @Column()<br>тип string<br>Слаг статті |
| title | @Column()<br>тип stringi<br>Заголовок статті |
| description | @Column({default: "})<br>тип string<br>Короткий опис статті |
| image | @Column({default: "})<br>тип string<br>Головне зображення статті |
| created | @Column({ type: 'timestamp', default: () =><br>"CURRENT_TIMESTAMP"})<br>тип Date<br>Дата створення статті |
| updated | @Column({ type: 'timestamp', default: () =><br>"CURRENT_TIMESTAMP"})<br>тип Date<br>Дата оновлення статті |

| | |
|---|---|
| tagList | @Column('simple-array')<br>тип string[]<br>Список тегів статті |
| author | @ManyToOne(type => UserEntity, user =><br>user.articles)<br>тип UserEntity<br>Автор статті |
| body | @Column({default: ''})<br>тип string<br>Тіло статті |
| exercises | @ManyToMany(type => ExerciseEntity, { eager:<br>true })<br>@JoinTable()<br>тип ExerciseEntity[];<br>Вправи які є в статті |
| comments | @OneToMany(type => Comment, comment =><br>comment.article, {eager: true})<br>@JoinColumn()<br>тип Comment[];<br>Коментарі до статті |
| favoriteCount | @Column({default: 0})<br>тип number;<br>Кількість вподобань |

4.2.1 Створення статті

Для створення статті необхідно надіслати запит типу POST за адресою /api/articles. Запит повинен містити заголовок Autorization з jwt токеном користувача, який є автором даної статті, а також саму статтю з наступними полями: заголовок, короткий опис, тіло статті, головна картинка статті, теги та вправи у форматі json. Детальніше про поля, їх типи та обов’язковість описано в файлі create-article.dto.ts з використанням патерну проектування DTO(data transfer object). Після валідації всіх полів на наявність та відповідність типу, створюється

екземпляр ArticleEntity та зберігається до бази даних. Якщо збереження виконано стаття буде додана до списку статей автора та повернена користувачу у форматі json.

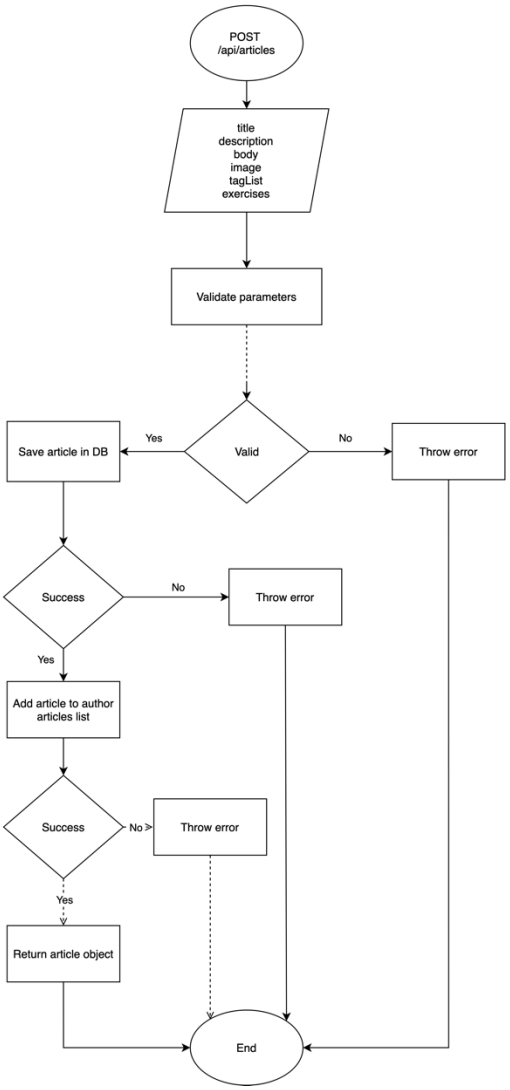



Рисунок 4.2.2 – Схема створення статті

### 4.2.2 Редагування та видалення статті

Редагування статті відбувається після отримання запиту типу PUT за даресою /api/article/:slug, де :slug – слаг статті. Редагувати статтю може тільки автор, тож необхідно його ідентифікувати за допомогою заголовку Autorization з jwt токеном користувача. Запит повинен містити ті поля статті, які необхідно оновити. Стаття

буде знайдена в базі по слагу, поля зі запиту буду оновлені і збережені в базі даних. Результатом редагування буде повернення об’єкту статті у форматі json користувачу.

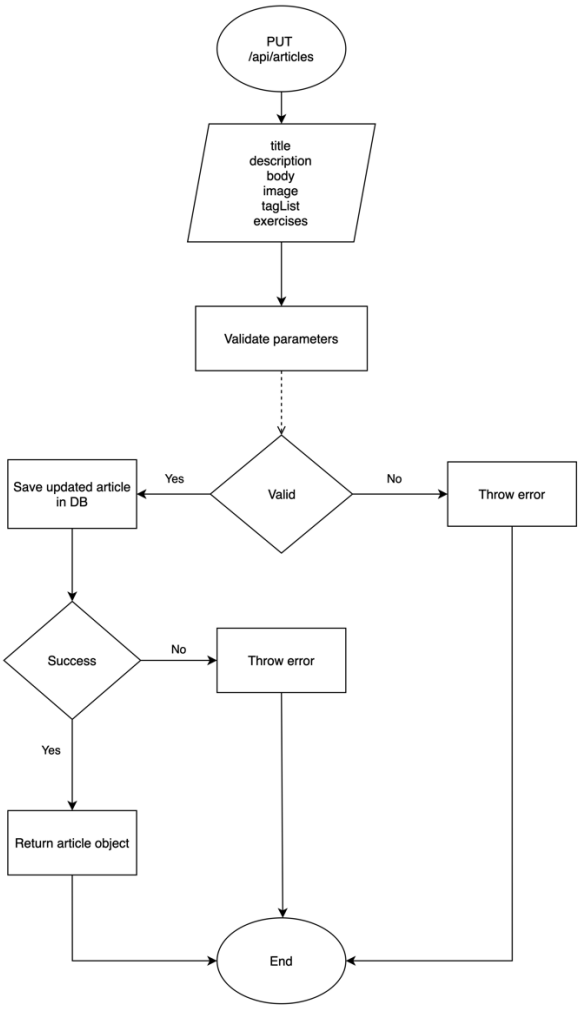



Рисунок 4.2.3 – Схема редагування статті

Видалення статті відбувається після отримання запиту типу DELETE за даресою /api/article/:slug, де :slug - слаг статті. Якщо стаття за вказаним в параметрах слагом буде знайдена в базі – вона буде видалена. Результатом успішного виконання буде повернуто повідомлення про успіх, або помилку, відповідно.

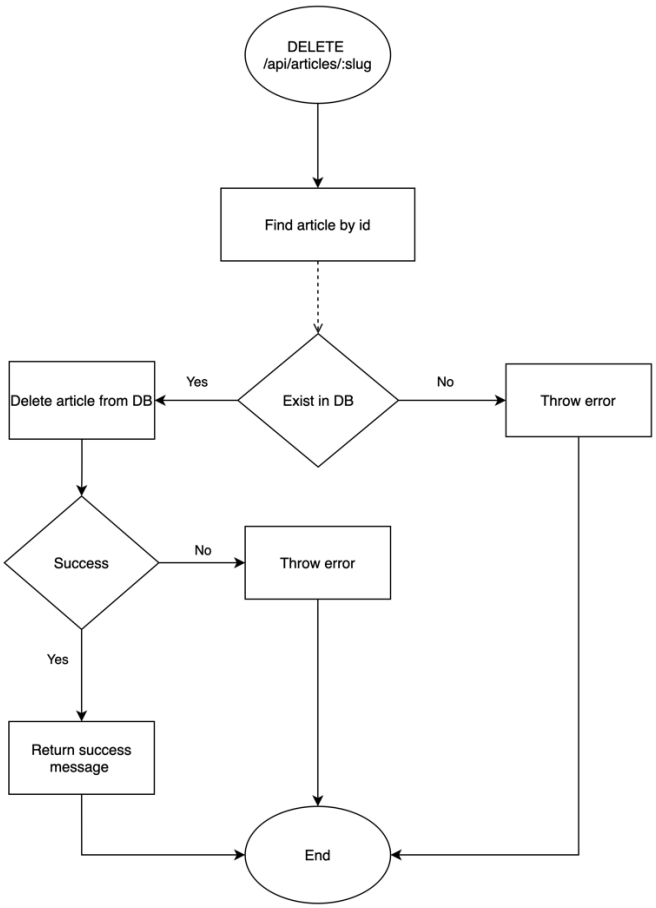



Рисунок 4.2.4 – Схема видалення. статті

### 4.2.3 Додання та видалення коментарів до статті

Кожна стаття може мати коментарі від авторизованих користувачів застосунку. Коментар додається до статті після отримання запиту типу POST за адресом /api/article/:slug/commets, де :slug - слаг статті. Так як коментарі можуть додавати лише авторизовані користувачі, необхідно перевірити авторизованого користувача по наявному в заголовку Autorization jwt токену користувача, якщо користувач авторизований відбувається перехід до додання коментарю. В базі даних необідно знайти статтю по слагу до якої додається коментар, створюється екземпляр коментаря з create-comment.dto.ts, додається коментар до списку коментарів та хберігається в базі даних. Результатом виконання буде повернуто об’єкт статті зі списком коментарі, або повідомлення про помилку виконання запиту.

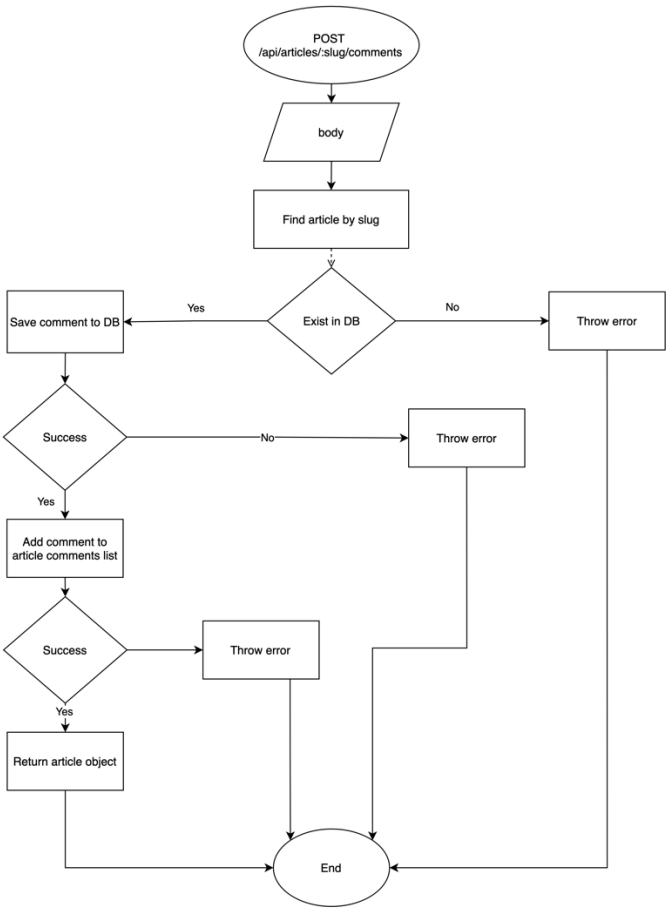



Рисунок 4.2.5 – Схема додання коментару

Видалення коментарю виконується за запитом типу DELETE за адресою /api/articles/:slug/comments/:id, де :slug – слаг статті до якої відноситься коментар, та :id – унікальний ідентифікатор коментарю. Упевнившись, що запит на видалення від автора коментарю переходимо до процесу видалення з бази даних. Знаходимо коментар по id та видаляємо з бази, одночасно оновлюємо список коментарів у статті. Результатом виконання буде повернення об’єкту статті в форматі json.

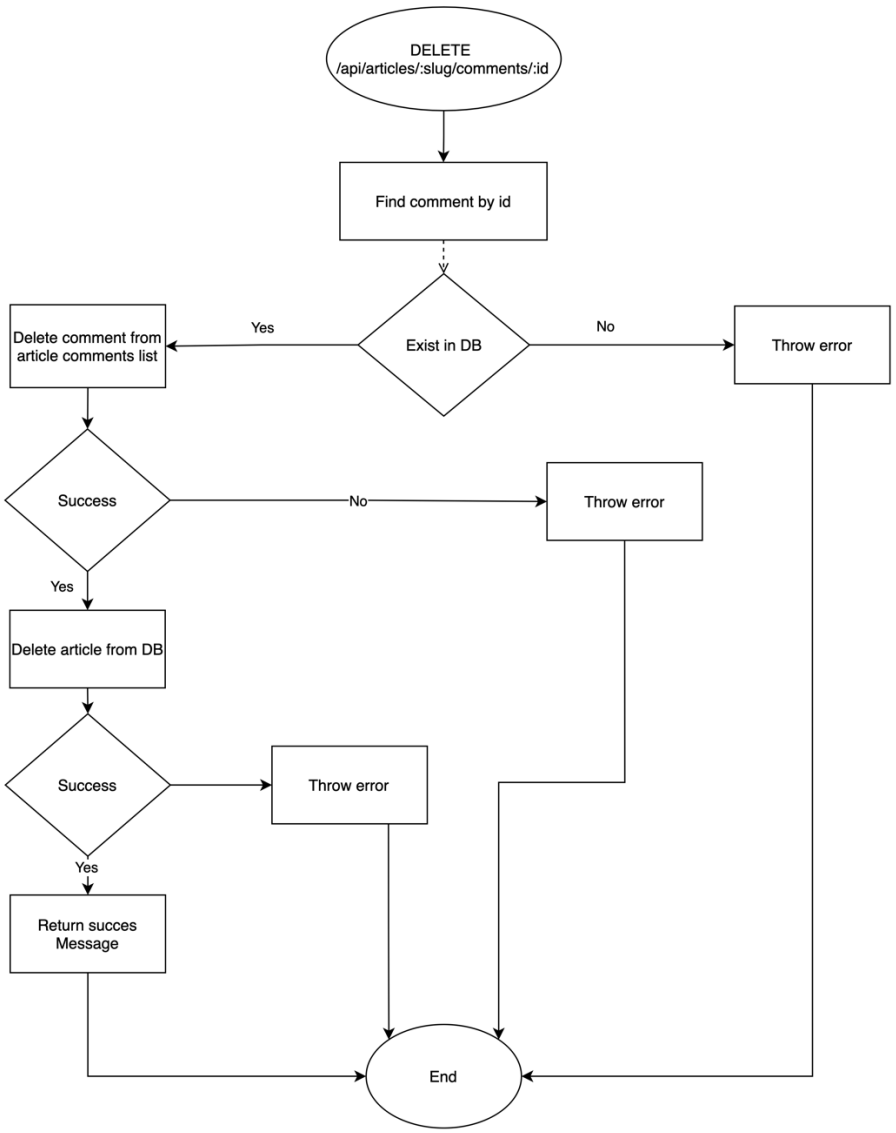



Рисунок 4.2.6 – Схема видалення коментарю

### 4.2.4 Додання та видалення вправ до статті

Автор статті може додавати наявні в базі даних вправи до своєї статті. Додання вправ до статті виконується за запитом типу POST на адресу /api/articles/:slug/exercises, де :slug – слаг статті до якої необхідно додати вправу. Запит містить унікальний ідентифікатор вправи, яку необхідно додати до статті та за цим ідентифікатором буде знайдено в базі даних вправу та додано до списку вправ статті, та повернуто оновлену статтю у вигляді json об’єкту. Якщо даної

вправи не буду в знайдено або виникне помилка при зберіганні змін – буде повернуто повідомлення з помилкою.

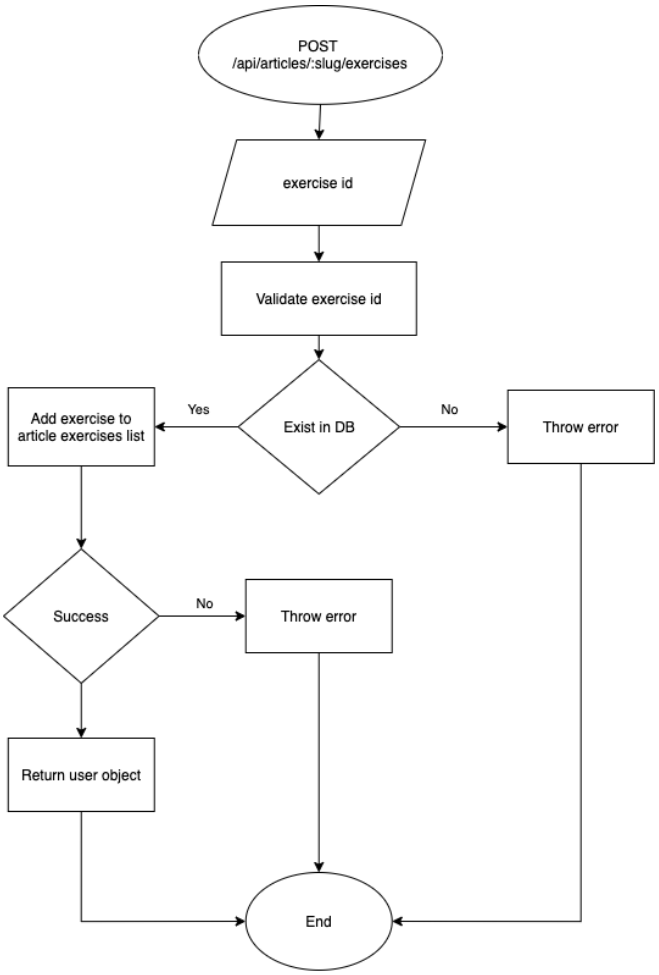



Рисунок 4.2.7 – Схема додання вправи до статті

Видалення доданої до статті вправи можливе за запитом типу DELETE на адресу /api/articles/:slug/exercises/:id, де :slug - слаг статті до якої додана вправа та :id – унікальний ідентифікатор вправи в застосунку. В базі даних буде знайдена стаття по слагу та видалено вправу, якщо видалення успішне буде повернуто оновлену статтю у вигляди json об’єкту, в інакшому випадку – повідомлення про помилку.

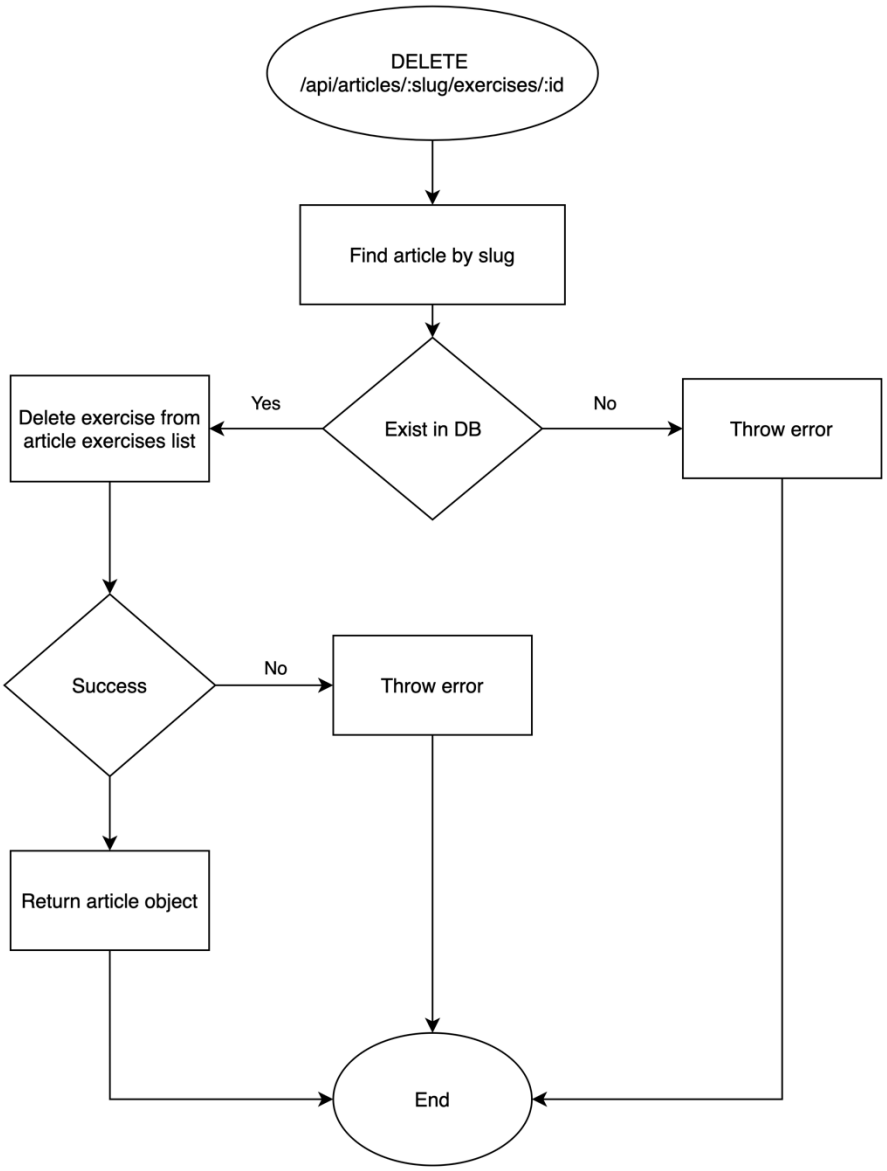



Рисунок 4.2.8 – Схема видалення вправи зі статті

### 4.2.5 Отримання списку статей на авторів яких підписаний користувач

Підписавшись на профілі інших користувачів застосунку, в своїй стрічні статей користувач буде отримувати тільки статті від відповідних користувачів. Додатково користувач може задавати фільтри для відображення, наприклад: чи містить стаття вправи, кількість статей на одній сторінці, сторінка та сортування. Отримати список статей можливо за запитом типу GET за на адресу /api/articles/feed. Після ідентифікації користувача за jwt токеном в заголовку запиту, відбираються користувачі на яких підписаний користувач та їх статті

відсортовуються за датою публікації або за параметром вказаним в параметрах запиту. Далі фільтруються відсортовані статті за фільтрами вказаними в параметрах. Результат повертається у вигляді масиву статей у вигляді json об’єктів.

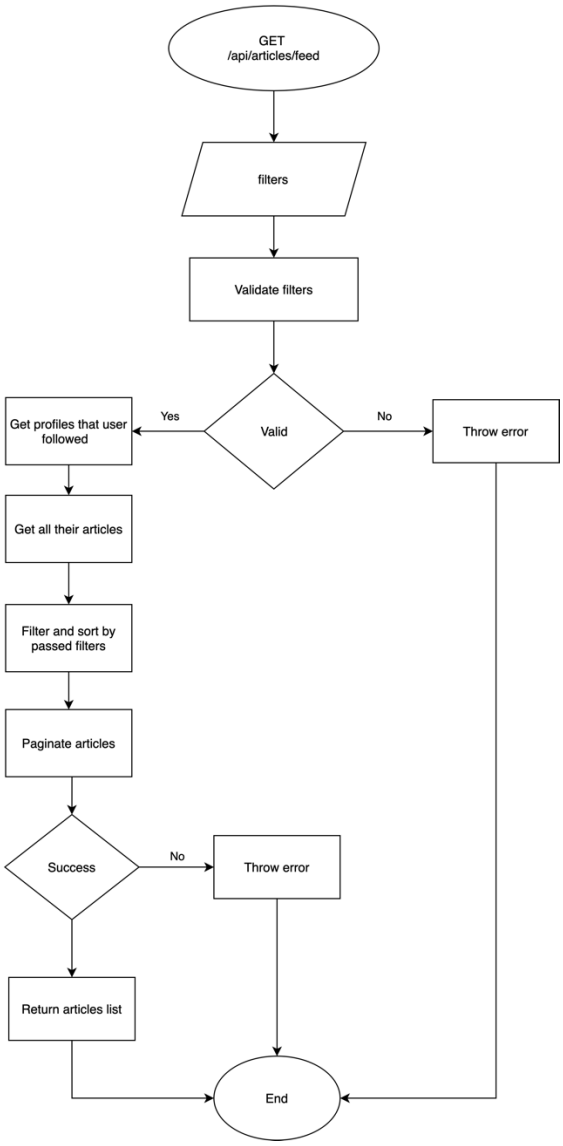
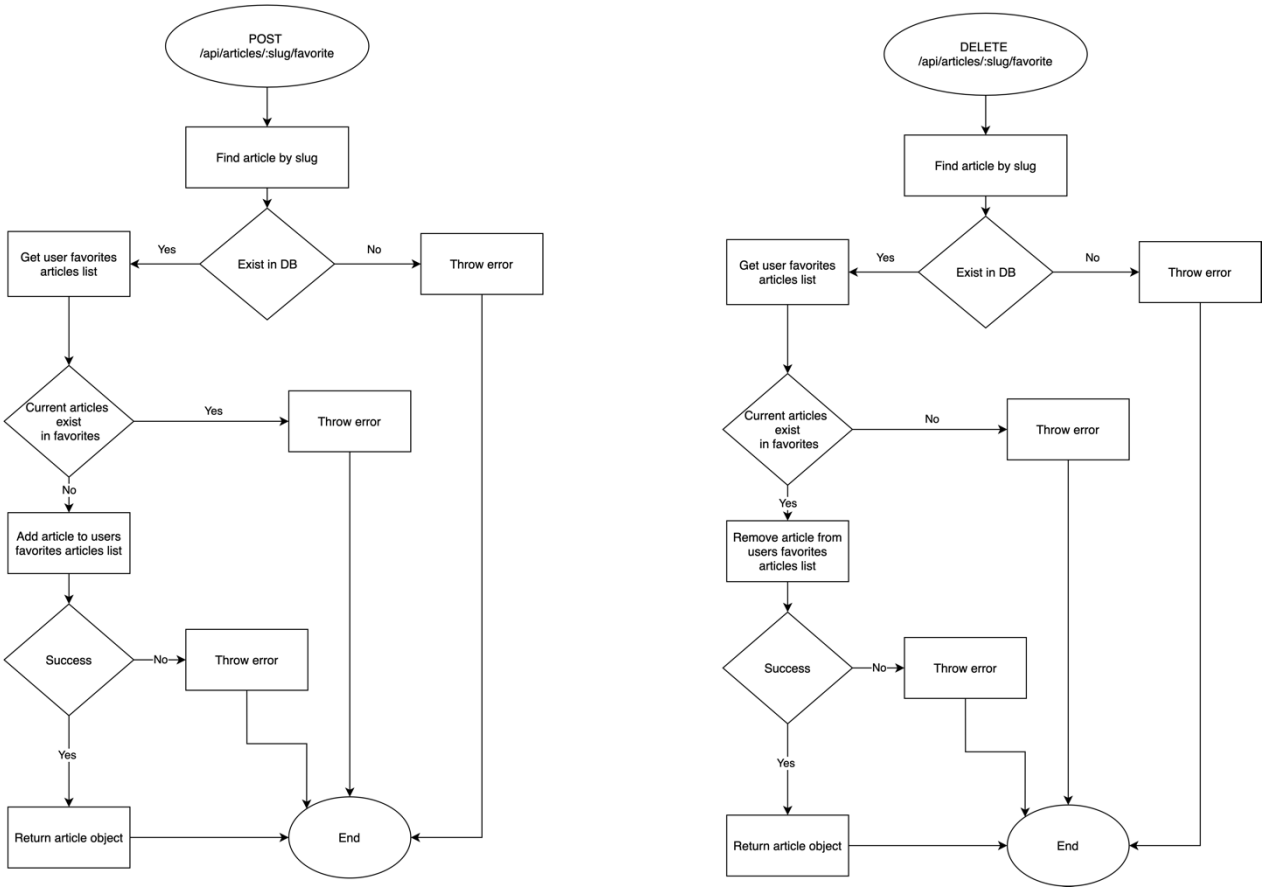



Рисунок 4.2.9 – Схема отримання списку статей

### 4.2.6 Додання та видалення статті зі списку вподобаних статей

Кожна стаття містить кількість вподобань від користувачів, також в базі даних у кожного користувача є перелік вподобаних статей. Стаття може отримати тільки одне вподобання від одного користувача, таким чином не буде проблем з накручуванням кількості вподобань статті. Додання статті до вподобаних

відбувається за запитом типу POST на адресу /api/articles/:slug/favorite, де :slug – слаг статті. Користувач визначається за jwt токеном, яких міститься в заголовку запиту, після чого в базі даних знаходиться стаття за слагом та збільшується кількість вподобань на 1, одночасно додається дана стаття до списку вподобаних статей користувача. Видалення статті зі списку вподобань відбувається аналогічним чином окрім типу запиту, він повинен бути DELETE. Результатом успішного виконання запиту є повернення оновленої статті в форматі json або повідомлення про помилку.




а) б)

Рисунок 4.2.10 – Алгоритм додавання (а) та видалення (б) статей із списку вподобань

## 4.3 Вправи

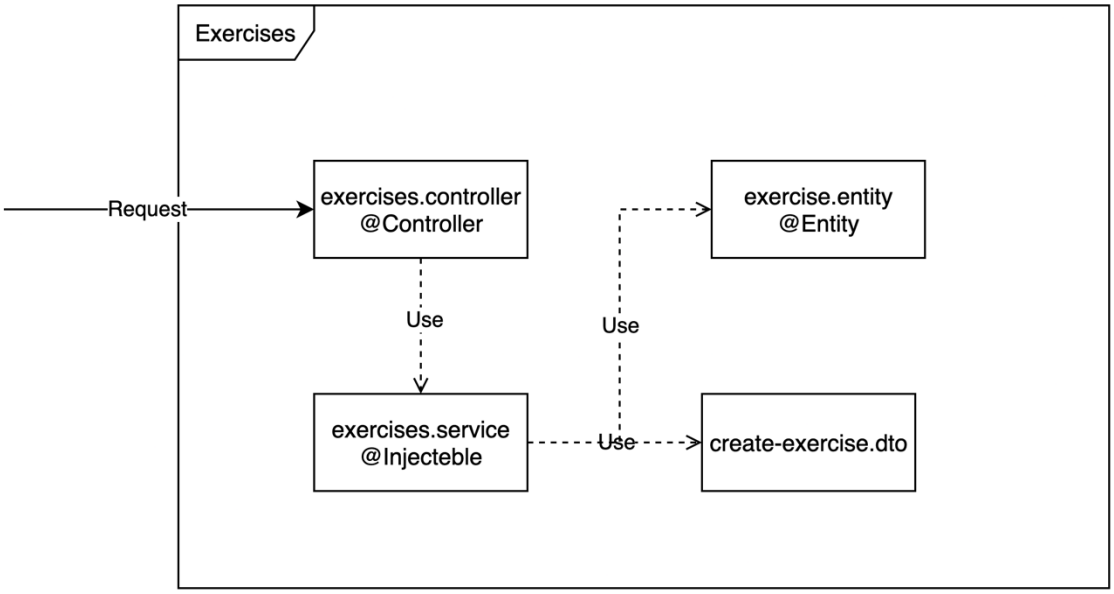



Рисунок 4.3.1 – Схема модулю вправ

Таблиця 4.2.4 – Методи класу ExercisesController

| Методи | Опис |
|---|---|
| findAll | Отримання всіх вправ. Виконується при отриманні GET запросу за адресою /api/exercises, повертає виклик методу ExerciseService.findAll |
| findOne | Отримання вправи по параметру slug. Виконується при отриманні GET запросу за адресою /api/exercises/:slug, повертає виклик методу ExerciseService.findAll |
| create | Створення вправи. Виконується при отриманні POST запросу за адресою /api/exercises повертає виклик методу ExerciseService.create |
| update | Оновлення вправи. Виконується при отриманні PUT запросу за адресою /api/exercises/:slug повертає виклик методу ExerciseService.update |

| | |
|---|---|
| delete | Видалення вправи. Виконується при отриманні DELETE запросу за адресою /api/exercises:slug повертає виклик методу ExerciseService.delete |

Таблиця 4.2.5 – Методи класу ExercisesService

| Методи | Опис |
|---|---|
| findAll | Отримання списку вправ з бази даних, які відповідають параметрам: теги, автор, кількість та сторінка. |
| findFeed | Отримання списку вправ з бази даних, на автори яких користувач підписаний. |
| findOne | Отримання вправи з бази даних за слагом. |
| create | Додання вправи до бази даних. |
| update | Оновлення вправи в базі даних. |
| delete | Видалення вправи з бази данних. |
| slugify | Створення слагу вправи з назви та чисел. |

Таблиця 4.2.6 – Поля класу ExercisesEntity

| Поле | Опис |
|---|---|
| id | @PrimaryGeneratedColumn()<br>тип number<br>Ідентифікатор вправи |
| slug | @Column()<br>тип string<br>Слаг вправи |
| title | @Column()<br>тип stringi<br>Заголовок вправи |
| description | @Column({default: ''})<br>тип string<br>Короткий опис вправи |
| image | @Column({default: ''})<br>тип string<br>Зображення вправи |
| created | @Column({ type: 'timestamp', default: () => "CURRENT_TIMESTAMP"})<br>тип Date<br>Дата створення вправи |

| | |
|---|---|
| updated | @Column({ type: 'timestamp', default: () => "CURRENT_TIMESTAMP"})<br>тип Date<br>Дата оновлення вправи |
| tagList | @Column('simple-array')<br>тип string[]<br>Список тегів вправи |
| author | @ManyToOne(type => UserEntity, user => user.exercises)<br>тип UserEntity<br>Автор вправи |

### 4.3.1 Створення вправи

Користувачів застосунку можуть створювати вправи, якщо необхідних вправ не має в базі даних. Для створення вправи необхідно надіслати запит на адресу /api/exercises, типу POST та в тілі запиту необхідно передати дані для створення вправи, а саме: назву вправи, зображення, список тегів та повний опис. Після перевірки даних, вправа буде додана до бази даних та до списку вправ користувача. Результатом успішного виконання буде повернуто об’єкт вправи у форматі json або повідомлення про помилку.

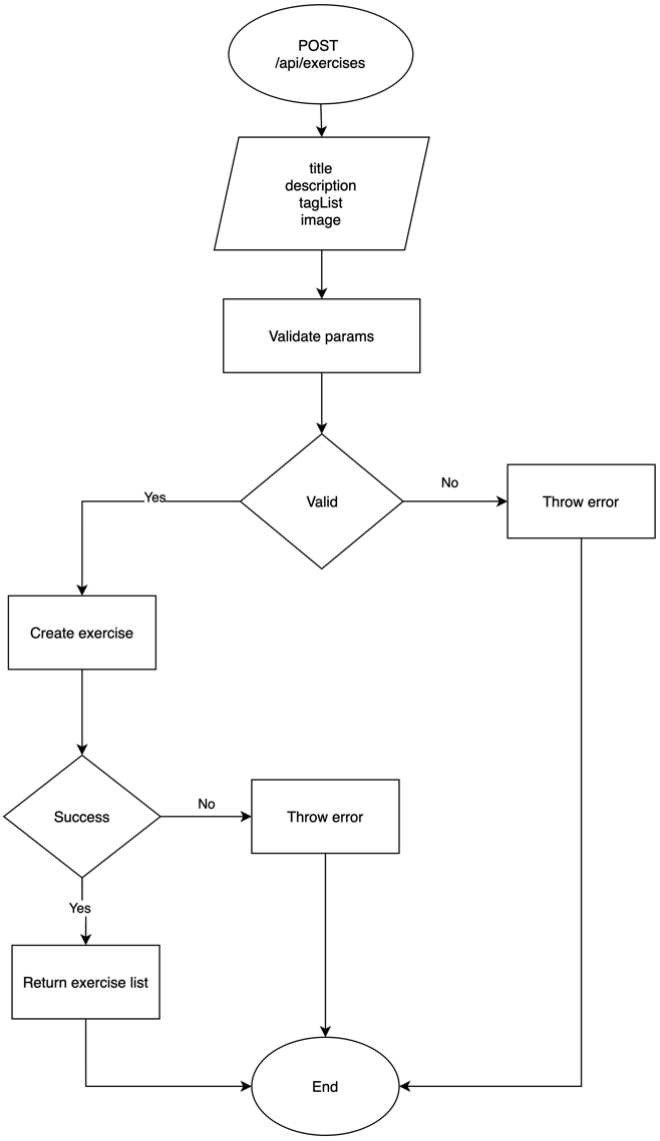



Рисунок 4.3.2 – Схема створення вправи

### 4.3.2 Редагування та видалення вправ

Редагування вправ відбувається за запитом типу PUT на адресу /api/exercises/:slug, де :slug – слаг вправи. Тіло запиту повинно містити поля вправи, які необхідно оновити. Після перевірки параметрів на відповідність буде знайдено вправу в базі даних, якщо така вправа наявна відбудеться оновлення та збереження, а результатом запиту буде повернуто об’єкт оновленої вправи у форматі json. Якщо вправа буде відсутня у базі даних, результатом запиту буде повернуто повідомлення про помилку редагування вправи.

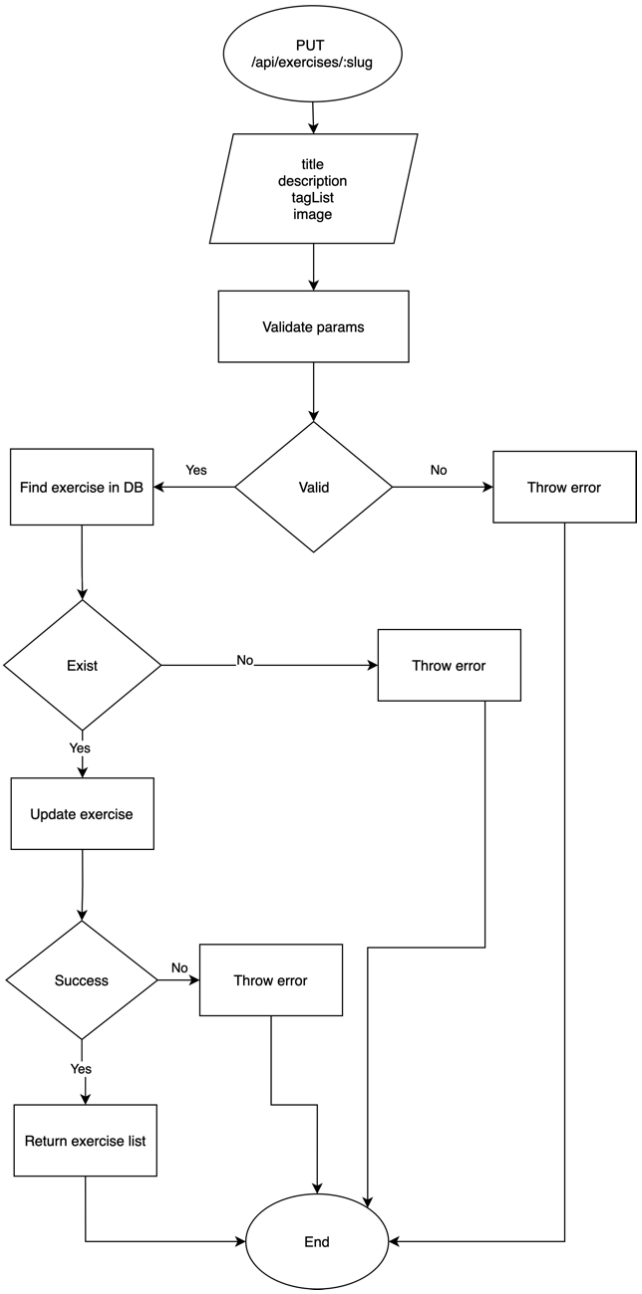



Рисунок 4.3.3 – Схема редагування вправи

Видалення вправ з бази даних застосунку відбувається за запитом типу DELETE на адресу /api/exercises/:slug, де :slug – слаг вправи для видалення. Після отримання запиту відбувається пошук вправи в базі даних, яку необхідно видалити. Якщо вправа знайдена, вона буде видалена з базі даних та буде оновлено список вправ користувача, результатом виконання запиту буде повернуто повідомлення про успішне видалення. В іншому випадку буде повернуто повідомлення про помилку видалення вправи.

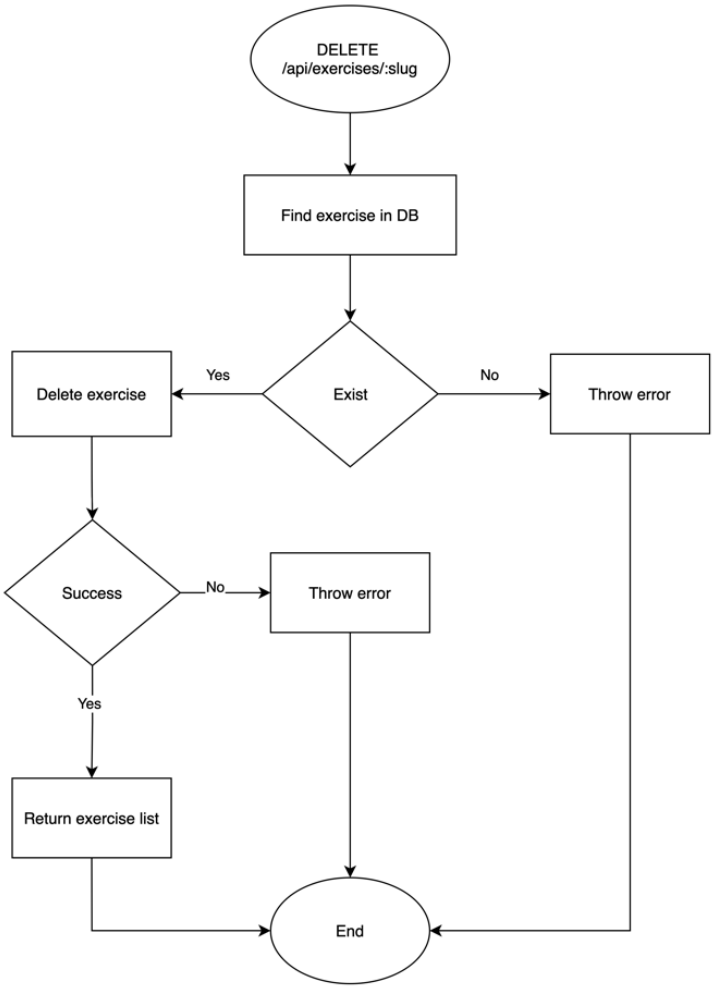



Рисунок 4.3.4 – Схема видалення вправи

## 4.4 Дієти

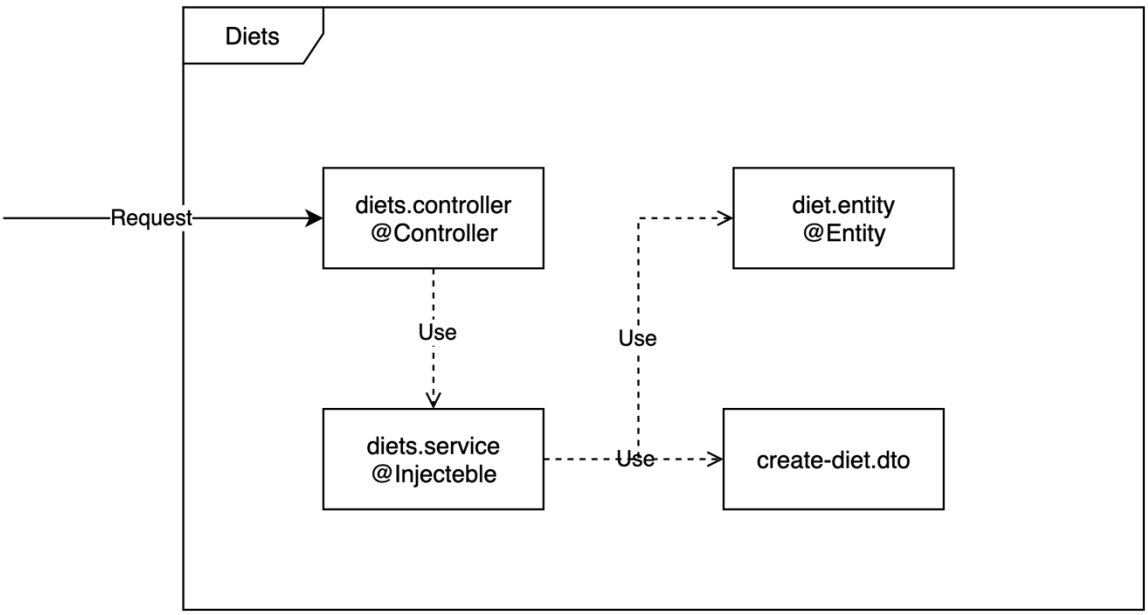



Рисунок 4.4.1 – Схема модулю дієт

Таблиця 4.4.1 – Методи класу DietsController

| Методи | Опис |
|---|---|
| findAll | Отримання спику всіх статей. Виконується при отриманні GET запросу за адресою /api/diets, повертає виклик методу DiestService.findAll |
| findOne | Отримання дієти по параметру slug. Виконується при отриманні GET запросу за адресою /api/diet/:slug, повертає виклик методу DietService.findAll |
| create | Створення дієти. Виконується при отриманні POST запросу за адресою /api/diets повертає виклик методу DietService.create |
| update | Оновлення дієти. Виконується при отриманні PUT запросу за адресою /api/diets/:slug повертає виклик методу DietService.update |
| delete | Видалення дієти. Виконується при отриманні DELETE запросу за адресою /api/diets/:slug повертає виклик методу DietService.delete |

Таблиця 4.4.2 – Методи класу DietsServices

| Методи | Опис |
|---|---|
| findAll | Отримання списку всії дієт з бази даних. |
| findFeed | Отримання списку статей з бази даних, на автори яких користувач підписаний. |
| findOne | Отримання дієти з бази даних за слагом. |
| create | Додання дієти до бази даних. |
| update | Оновлення дієти в базі даних. |
| delete | Видалення дієти з бази данних. |
| slugify | Створення слагу дієти з назви та чисел. |

Таблиця 4.4.3 – Поля класу DietEntity

| Поле | Опис |
|---|---|
| id | @PrimaryGeneratedColumn()<br>тип number<br>Ідентифікатор дієти |
| slug | @Column()<br>тип string<br>Згенерований слаг дієти |
| title | @Column()<br>тип stringi<br>Назва дієти |
| description | @Column({default: ''})<br>тип string<br>Короткий опис дієти |
| image | @Column({default: ''})<br>тип string<br>Головне зображення дієти |
| created | @Column({ type: 'timestamp', default: () =><br>"CURRENT_TIMESTAMP"})<br>тип Date<br>Дата створення дієти |
| updated | @Column({ type: 'timestamp', default: () =><br>"CURRENT_TIMESTAMP"})<br>тип Date<br>Дата оновлення дієти |
| tagList | @Column('simple-array')<br>тип string[]<br>Список тегів дієти |

| | |
|---|---|
| author | @ManyToOne(type => UserEntity, user => user.articles)<br>тип UserEntity<br>Автор дієти |
| type | @Column({default: Types.Free})<br>тип Types<br>Тип вправи(безкоштовна або платна) |
| price | @Column({default: 0})<br>тип number<br>Ціна вправи |

### 4.4.1 Створення дієти

Для створення дієти необхідно надіслати запит на адресу /api/diets, типу POST та в тілі запиту необхідно передати дані для створення дієти, а саме: назву дієти, зображення, список тегів, повний опис, тип дієти(безкоштовна чи платна) та ціну, якщо платна. Після перевірки даних, дієта буде додана до бази даних та до списку дієт користувача. Результатом успішного виконання буде повернуто об’єкт дієти у форматі json або повідомлення про помилку.

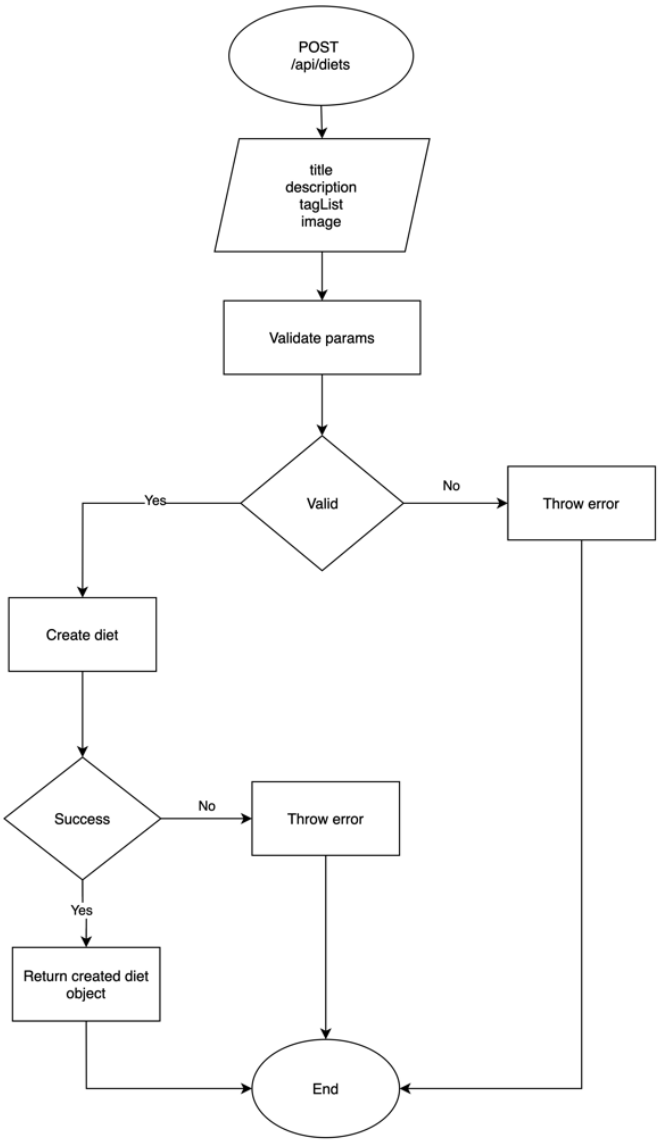



Рисунок 4.4.2 – Схема створення дієти

### 4.4.2. Редагування та видалення дієти

Редагування дієти відбувається за запитом типу PUT на адресу /api/diets/:slug, де :slug – слаг дієти. Тіло запиту повинно містити поля дієти, які необхідно оновити. Після перевірки параметрів на відповідність буде знайдено дієту в базі даних, якщо така дієта наявна відбудеться оновлення та збереження, а результатом запиту буде повернуто об’єкт оновленої дієти у форматі json. Якщо дієта буде відсутня у базі даних, результатом запиту буде повернуто повідомлення про помилку редагування дієти.

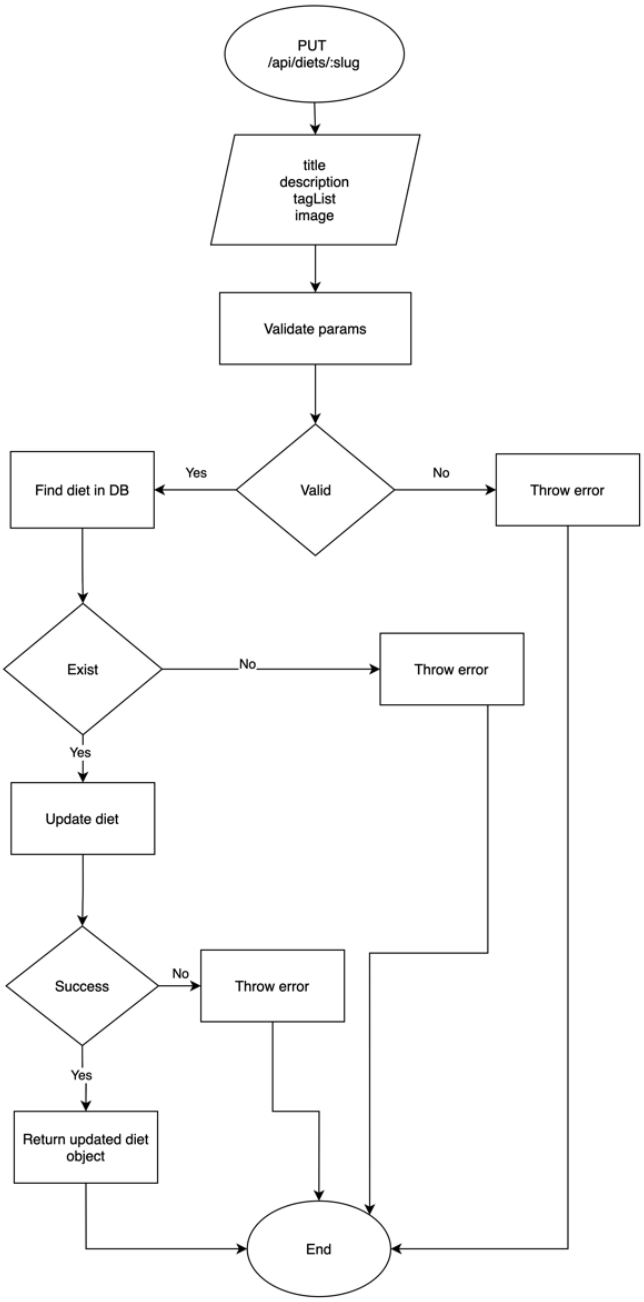



Рисунок 4.4.3 – Схема редагування дієти

Видалення дієти з бази даних застосунку відбувається за запитом типу DELETE на адресу /api/diets/:slug, де :slug – слаг дієти. Після отримання запиту відбувається пошук дієти в базі даних, яку необхідно видалити. Якщо дієта знайдена, вона буде видалена з базі даних та буде оновлено список дієт користувача, результатом виконання запиту буде повернуто повідомлення про успішне видалення. В іншому випадку буде повернуто повідомлення про помилку видалення.

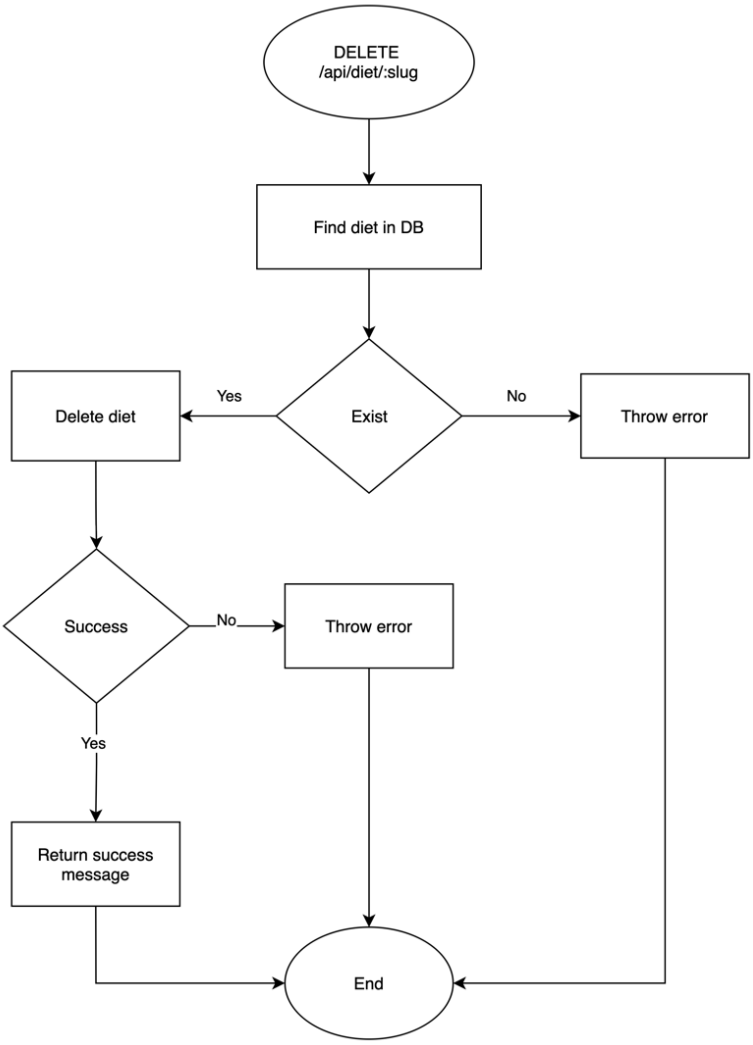



Рисунок 4.4.4 – Схема видалення дієти

## 4.5 Профіль

Кожен користувач застосунку має свій профіль, який буде відображатися іншим користувачам застосунку. Профіль користувача містить лише поля: ім’я, прізвище, зображення, коротку інформацію, ім’я користувача в застосунку з його профілю, та доповнюється інформацією про підписки. Також додає функціонал підписки та відписки від користувачів, що дає змогу відстежувати активність користувачів.

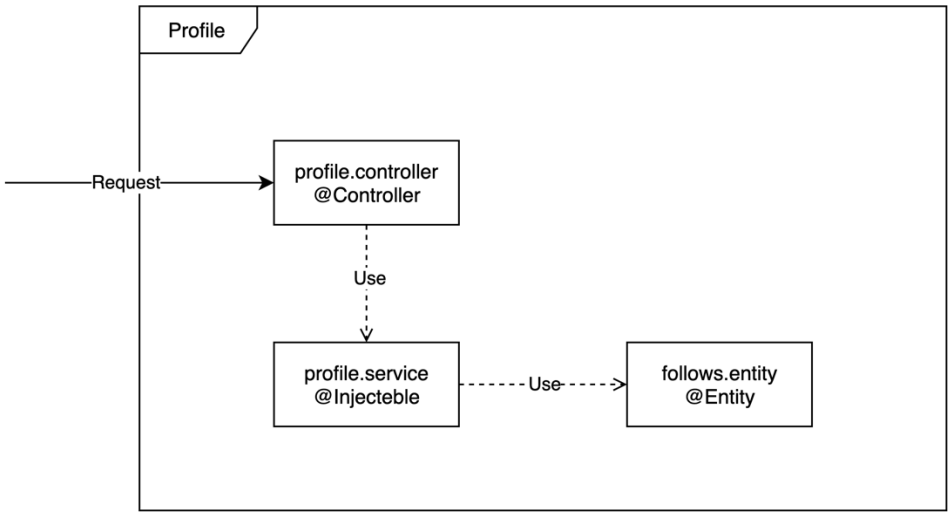



Рисунок 4.5.1 – Схема модулю профіль

Таблиця 4.5.1 – Методи класу ProfileController

| Методи | Опис |
|---|---|
| getProfile | Отримання всіх вправ. Виконується при отриманні GET запросу за адресою /api/exercises, повертає виклик методу ExerciseService.findAll |
| follow | Отримання вправи по параметру slug. Виконується при отриманні GET запросу за адресою /api/exercises/:slug, повертає виклик методу ExerciseService.findAll |
| unFollow | Створення вправи. Виконується при отриманні POST запросу за адресою /api/exercises повертає виклик методу ExerciseService.create |

Таблиця 4.5.2 – Методи класу ProfileServices

| Методи | Опис |
|---|---|
| findAll | Отримання списку всіх користувачів с бази даних. |
| findProfile | Отримання даних профілю користувача. |
| findOne | Отримання користувача з бази даних за унікальним ім'ям в застосунку. |

| | |
|---|---|
| follow | Підписка на профіль користувача та оновлення профілю. |
| unFollow | Відписка на профіль користувача та оновлення профілю. |

Таблиця 4.5.3 – Поля класу ProfileEntity

| Поле | Опис |
|---|---|
| id | @PrimaryGeneratedColumn()<br>тип number<br>Ідентифікатор вправи |
| followerId | @Column()<br>тип number<br>Ідентифікатор користувача що підписаний |
| followingId | @Column()<br>тип number<br>Ідентифікатор користувача на кого підписалися |

### 4.5.1 Отримання профілю користувача

Для отримання профілю користувача необхідно відправити запит типу GET на адресу /api/:username, де :username – унікальне ім’я користувача в застосунку. Після пошуку в базі даних користувача за його ім’ям, сформується результат з доступними іншим користувачам даними. Результатом успішного виконання операції буде повернуто об’єкт профілю користувача у форматі json, або повідомлення про помилку.

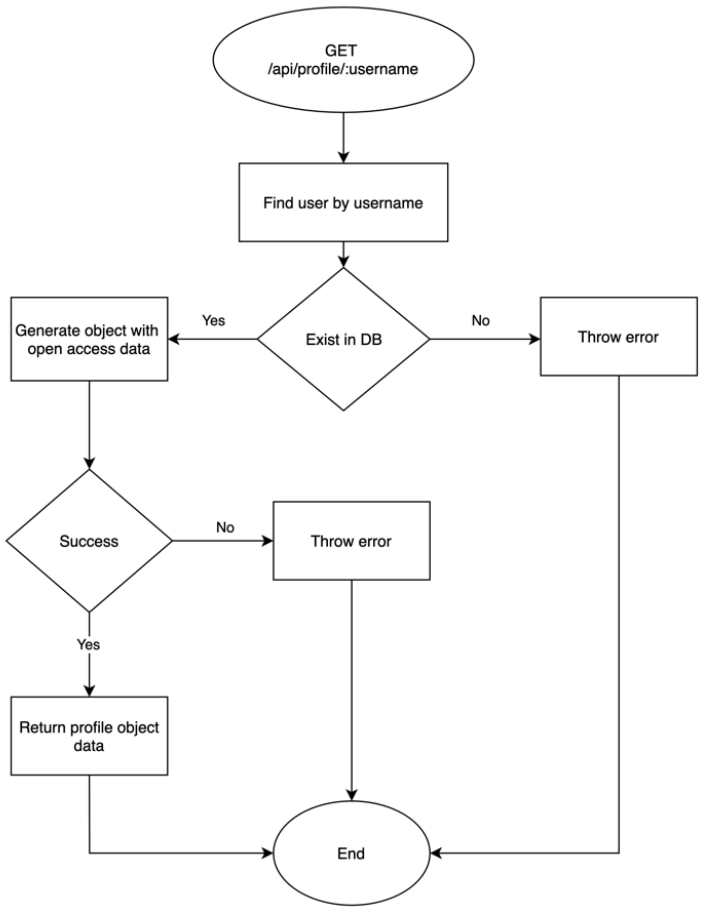



Рисунок 4.5.2 – Схема отримання профілю користувача

### 4.5.2 Підписка та відписка від профіль користувача

Користувачі застосунку можуть підписуватися на профіль зареєстрованих користувачів в застосунку. Підписка відбувається надсиланням запиту типу POST на адресу /api/profiles/:username/follow, де :username – ім’я користувача в застосунку на якого необхідно підписатися. Після отримання запиту відбудеться пошук користувача в базі даних за його ім’ям в застосунку, якщо такий користувач є та в списку підписаних на нього користувачів не має користувача, що хоче підписатися – необхідно додати його в список. Результатом успішного виконання запиту буде відправлено об’єкт профілю користувача з оновленим полем following зі значенням true, яке повідомляє про активну підписку на його профіль, в іншому випадку буде повернуто повідомлення про помилку.

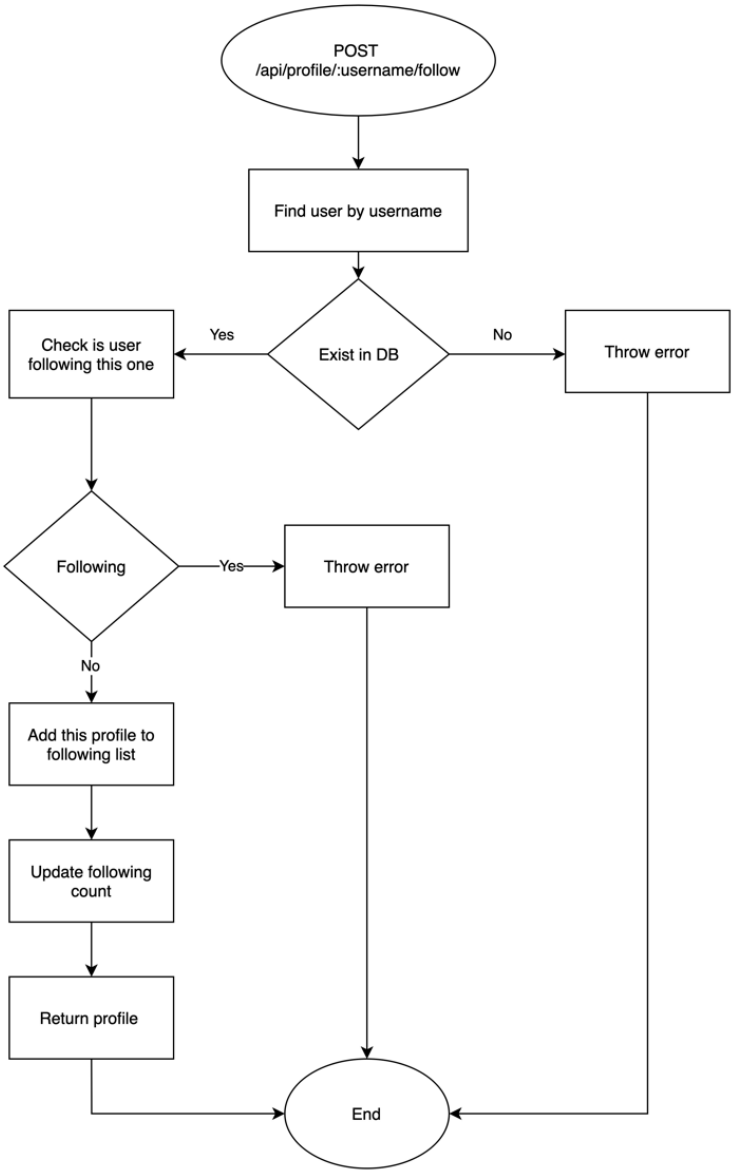



Рисунок 4.5.3 – Схема підписки від профілю користувача

Для відписки від профілю користувача необхідно надіслати запит типу DELETE на адресу /api/profiles/:username/follow, де :username – ім’я користувача в застосунку від якого необхідно відписатися. Після отримання запиту відбудеться пошук користувача в базі даних за його ім’ям в застосунку, якщо такий користувач є та в списку підписаних на нього користувачів є користувач, що хоче відписатися – необхідно видалити його. Результатом успішного виконання запиту буде відправлено об’єкт профілю користувача з оновленим полем following зі значенням false, яке повідомляє про активну підписку на його профіль, або повернуто повідомлення про помилку.

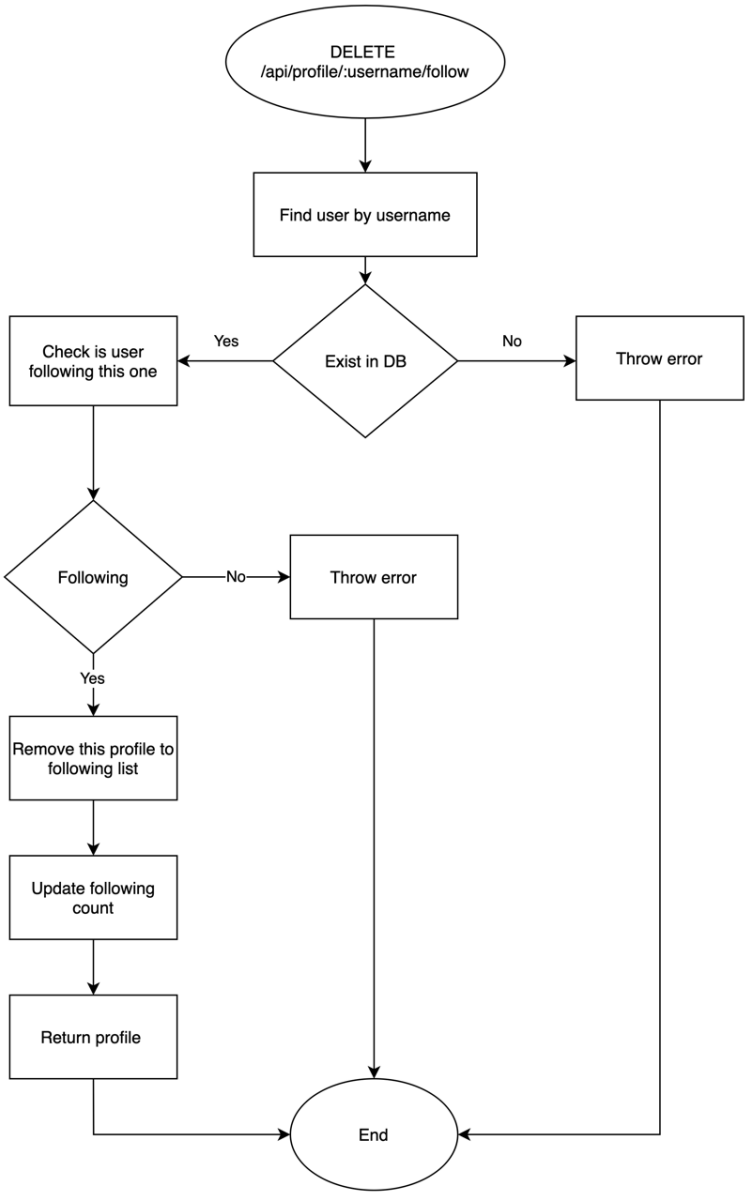



Рисунок 4.5.4 – Схема відписки від профілю користувача

## 4.6 Робота з зображеннями

Так як в розділі статті, вправи, дієти та профілі користувачів містять зображення, для їх завантаження та збереження було створення окремий модуль. Під час завантаження зображень користувачі можуть використовувати картинки різного розміру, якщо ж зображеннями малого розміру проблем не виникає то з великим розміром є проблема із зберіганням на сервері. Зі збільшенням кількості зображень – необхідно більше місця для їх зберігання, для вирішення даної проблеми було створено «воркера» для стиснення та зменшення розміру завантажених зображень.

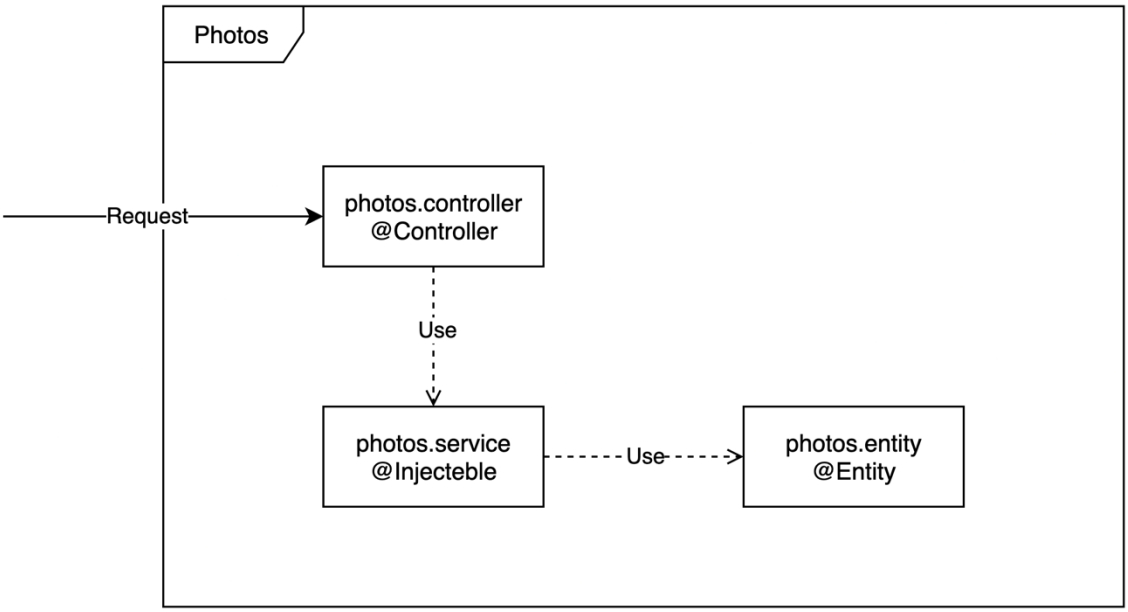



Рисунок 4.6.1 – Схема модулю зображень

Таблиця 4.6.1 – Методи класу PhotosController

| Методи | Опис |
|---|---|
| uploadPhoto | Завантаження зобрадення до серверу. Виконується при отриманні POST запросу за адресою /api/photos/upload, повертає виклик методу PhotosService.upload |

Таблиця 4.6.2 – Методи класу PhotosServices

| Методи | Опис |
|---|---|
| upload | Збереження зображення на сервер та збереження інформації про шлях до файлу та статус до бази даних |
| startWorker | Через заданий інтервал часу стискає зображення та зберігає в тому ж форматі |

Таблиця 4.6.3 – Методи класу PhotoEntity

| Поле | Опис |
|---|---|
| id | @PrimaryGeneratedColumn()<br>тип number<br>Ідентифікатор вправи |

| | |
|---|---|
| pathUncompressed | @Column()<br>тип string<br>Шлях до не стиснутого зображення |
| pathCompressed | @Column()<br>тип stringi<br>Шлях до стиснуго зображення |
| status | @Column({default: PhotoStatuses.Uncompressed})<br>тип PhotoStatuses<br>Статус збереженого зображення(стиснуто або не стиснуто) |

*Завантаження зображень*. Для завантаження зображення на сервер необхідно відправити запит типу POST на адресу /api/photos/upload та в тілі запиту повинен бути файл зображення (рис. 4.6.1).

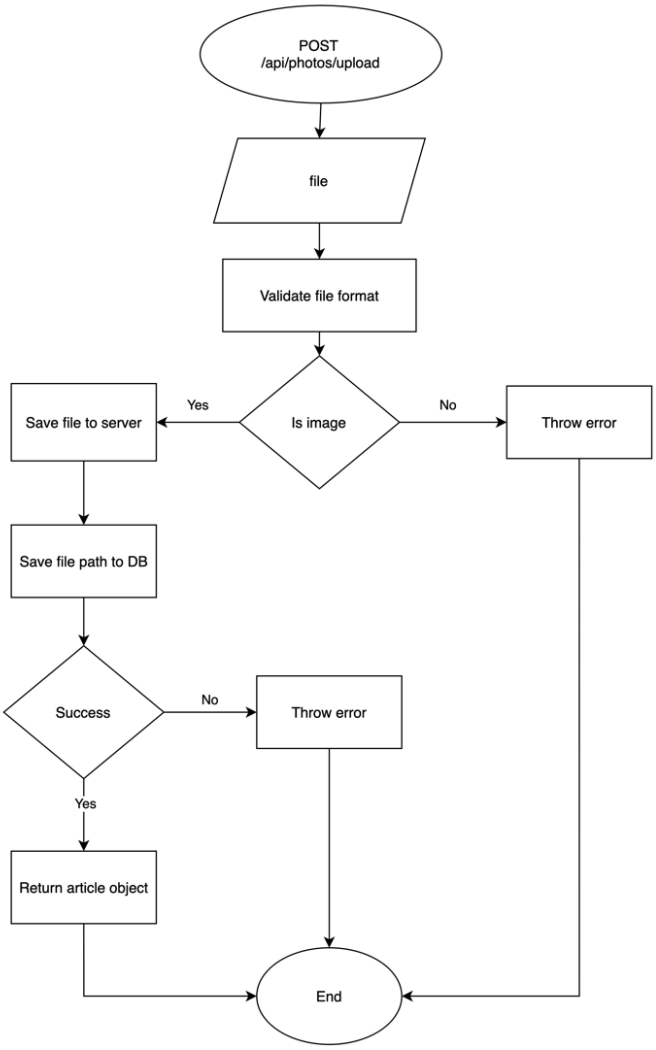



Рисунок 4.6.1 – Схема завантаження зображення

Після перевірки формату файлу на відповідність зображенню він буде збережений на сервері та до бази даних буде поміщено інформацію про шлях до зображення та статус (стиснуто або не стиснуто). Результатом виконя операції буде повернуто повідомлення про успішне завантаження, або повідомлення про помилку.

Висновки до розділу 4

Описано розробку основних частин серверної частини вебзастосунку, наведено схеми виконання обробки запитів, що надходять до серверу, описано методів та полів класів, що використовуються.

# 5 РОЗРОБКА КЛІЄНСЬКОЇ ЧАСТИНИ

## 5.1 Створення проекту

Для створення nuxtjs проекту необхідно встановити create-nuxt-app пакет, він дозволяє створити проект з певним набором додаткових пакетів(рис. 5.1.1)

```
❯ create-nuxt-app fitness-app-frontend

create-nuxt-app v3.6.0
✨  Generating Nuxt.js project in fitness-app-frontend
? Project name: fitness-app-frontend
? Programming language: JavaScript
? Package manager: Yarn
? UI framework: Buefy
? Nuxt.js modules: Axios - Promise based HTTP client, Progressive Web App (PWA)
? Linting tools: ESLint, Prettier
? Testing framework: Jest
? Rendering mode: Universal (SSR / SSG)
? Deployment target: Server (Node.js hosting)
? Development tools: jsconfig.json (Recommended for VS Code if you're not using
typescript)
? Continuous integration: GitHub Actions (GitHub only)
? What is your GitHub username? alexandroleinik
? Version control system: Git
```

Рисунок 5.1.1 – Налаштування проекту

Після вибору необхідних пакетів буде запущено процес завантаження та встановлення, в завершені операції буде створена папка з проектом.

## 5.2 Налаштування vuex

Багато компонентні веб-застосунки взаємодіють з даними, які можуть бути спільні або змінюватися в інших частинах застосунку і це викликає не зручності роботи з даними, щоб вирішити дану проблему використовуємо vuex. Vuex розбиваємо на різні сутності для зручності та простоти роботи.

Таблиця 5.1.1 – Сутності vuex

| Сутність vuex | Опис |
|---|---|
| user | Авторизований користувач |
| articles | Статі |
| exercises | Вправи |
| diets | Дієти |
| common | Загальні данні |
| profiles | Профілі користувачів |
| photos | Фото |
| feed | Стрічка статей та фільтрація |

## 5.3 Реєстрація та авторизація користувачів

Для реєстрації користувачів була створена сторінка з формою реєстрації, яка містить наступні поля: email, пароль, повтор пароля та вибір типу користувача(тренер чи спортсмен). Данна сторінка доступна за адресом /register. Після відправки форми на сервер на адресу /api/users, якщо реєстрація пройде успішно отримаємо профіль та jwt токен користувача, який буде збережено в vuex за допомогою nuxt-auth та буде використовуватися при запитах до серверу. Далі користувача буде переправлено на сторінку заповнення його персональних даних за адресом /register/next. Заповнивши форму користувачу необхідно натиснути кнопку «Продовжити», відправивши запит на оновлення даних профілю на адресу /api/user, якщо оновлення буде успішним відбудеться перехід на головну сторінку застосунку.

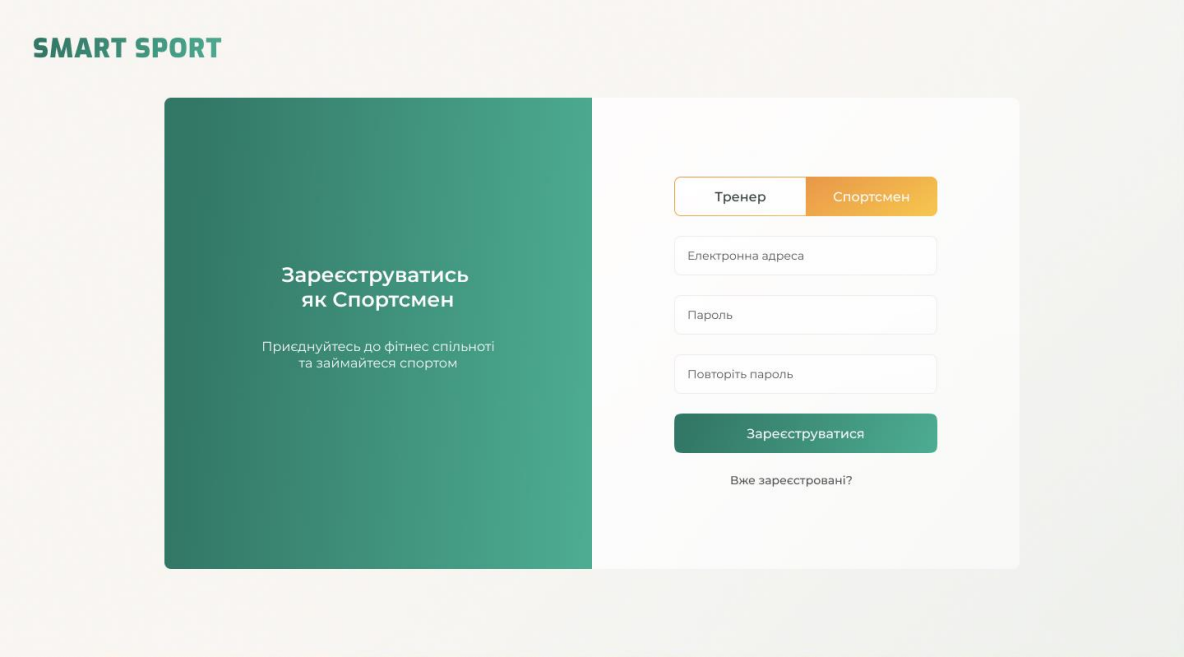



Рисунок 5.3.1 – Форма реєстрації «Спортсмена»

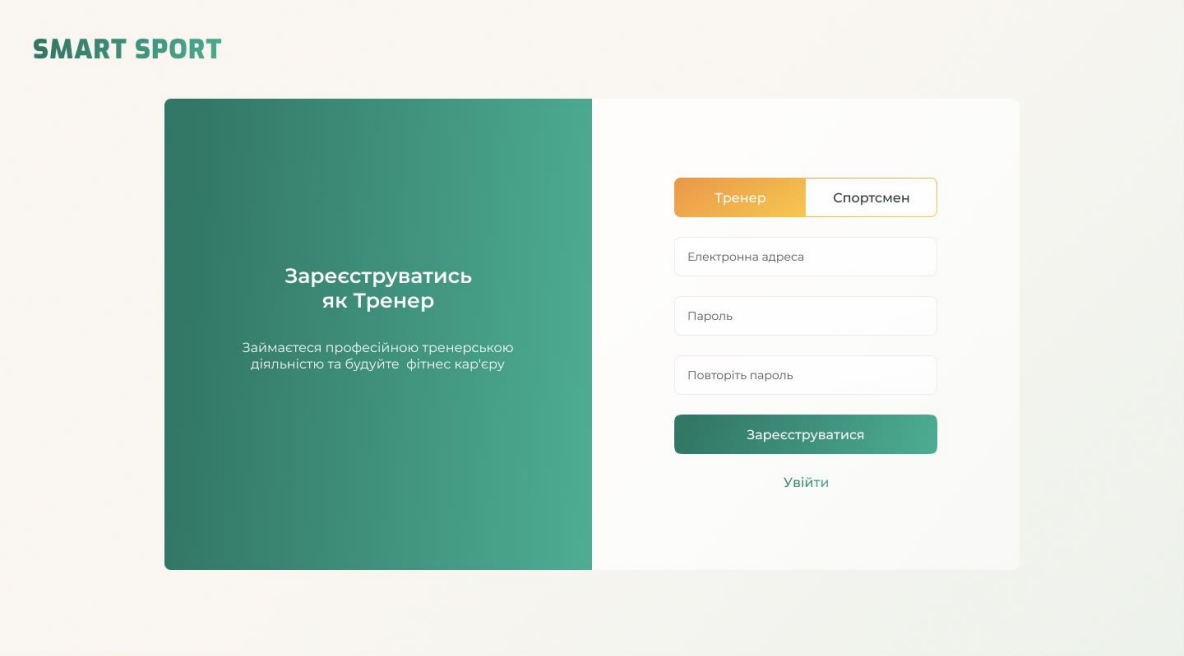



Рисунок 5.3.2 – Форма реєстрації «Тренера»

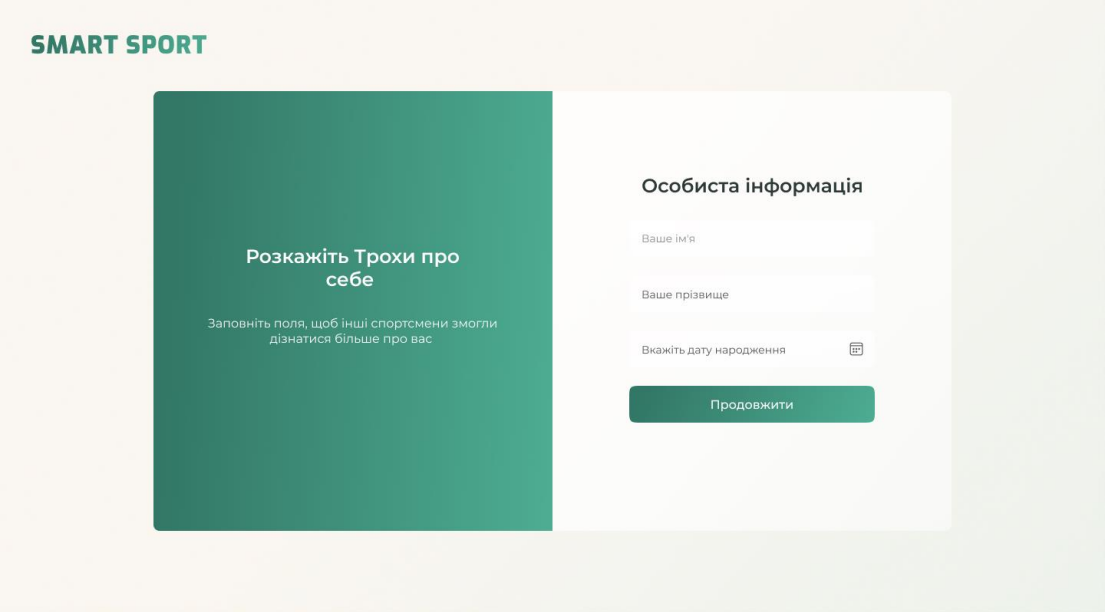



Рисунок 5.3.3 – Форма заповнення особистих даних під час реєстрації

Для повноцінної взаємодії користувача з веб-застосунком необхідно бути авторизованим в ньому. Для авторизації в застосунку необхідно натиснути на кнопку «Увійти» або при спробі скористуватися функціями, які не доступні не авторизованим користувачам буде виконано перехід на сторінку авторизації. Для контролю доступу до сторінок застосунку, які доступні лише авторизованим користувачам до кожної сторінки необхідно в параметр «auth» передати булеве значення true або false. Таким чином перед кожною змінною адреси буде спрацьовувати проміжне програмне забезпечення, яке перевіряє авторизацію користувача та вирішує виконати перехід або переправити на сторінку авторизації. На сторінці міститься форма з двома полями email та пароль від профілю користувача, після відправки форми на адресу /api/users/login/ буде повернуто згенерований jwt токен для доступу, який зберігається в localStorage браузеру.

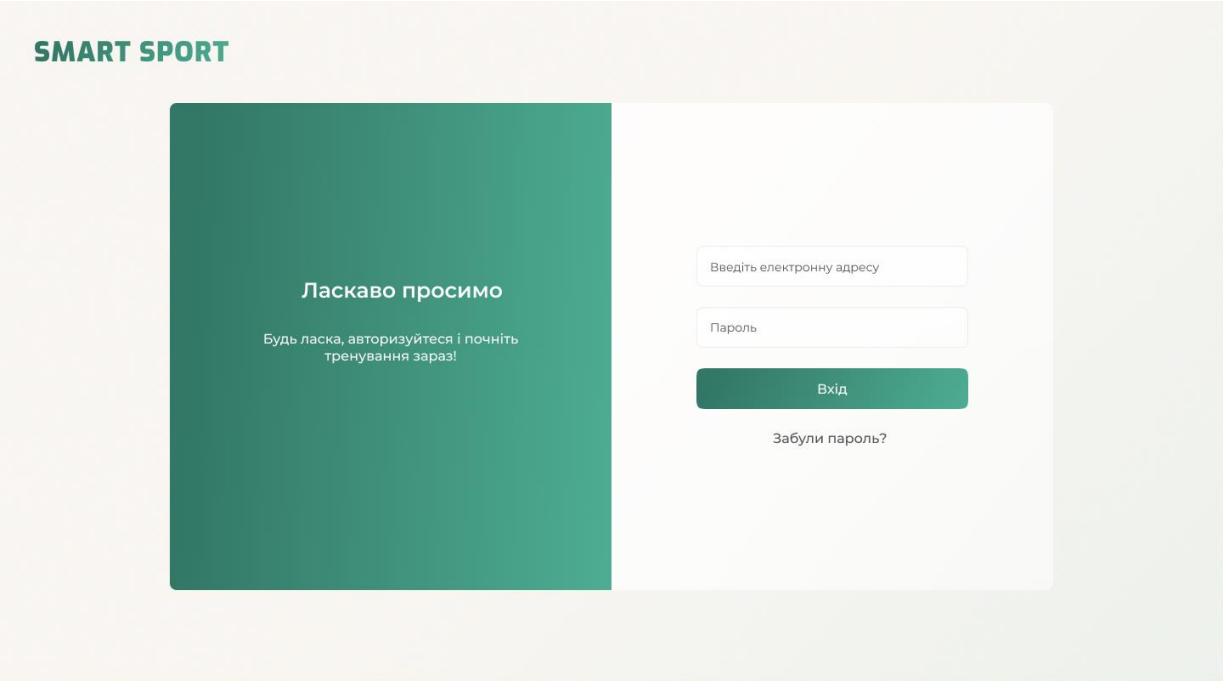



Рисунок 5.3.4 – Сторінка авторизації в застосунку

## 5.4 Редагування профілю користувача

Після реєстрації профіль користувача не заповнений додатковою інформацією про користувача. І виглядає порожнім, як показано на рис. 5.4.1.

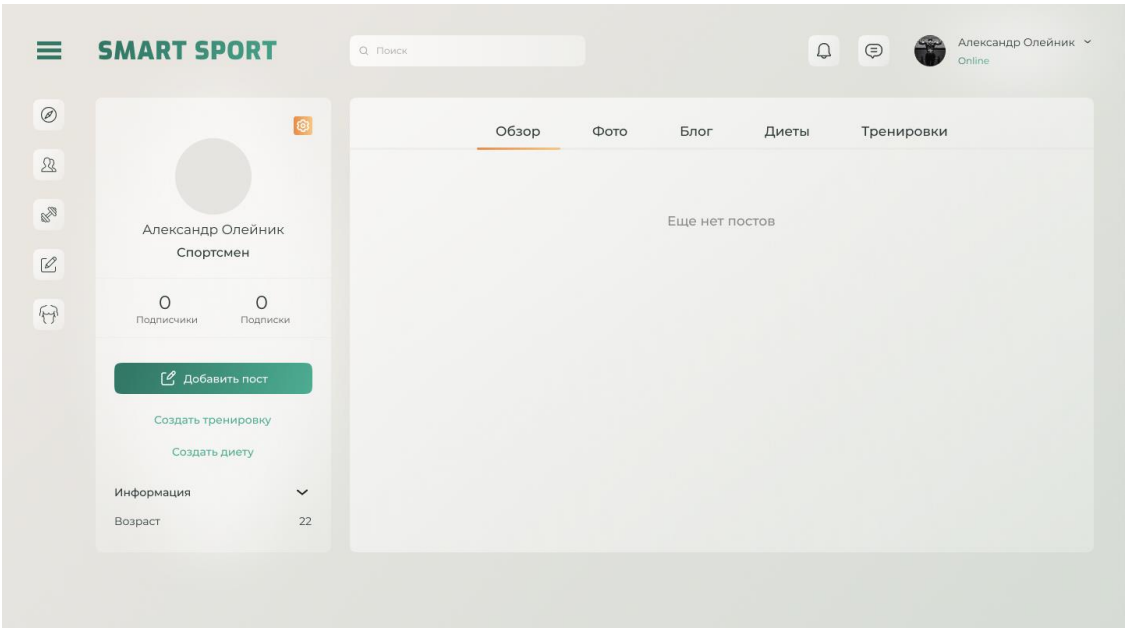



Рисунок 5.4.1 – Порожній профіль користувача

Для редагування даних профілю необхідно перейти на відповідну сторінку шляхом кліку на своє ім’я в хедері та на сторінці профілю натиснути на кнопку «Редагувати». Підчас перехід на сторінку запитом на адресу /api/users/me отримується актуальна інформацію профілю та необхідно заповнити поля форми відповідними даними.

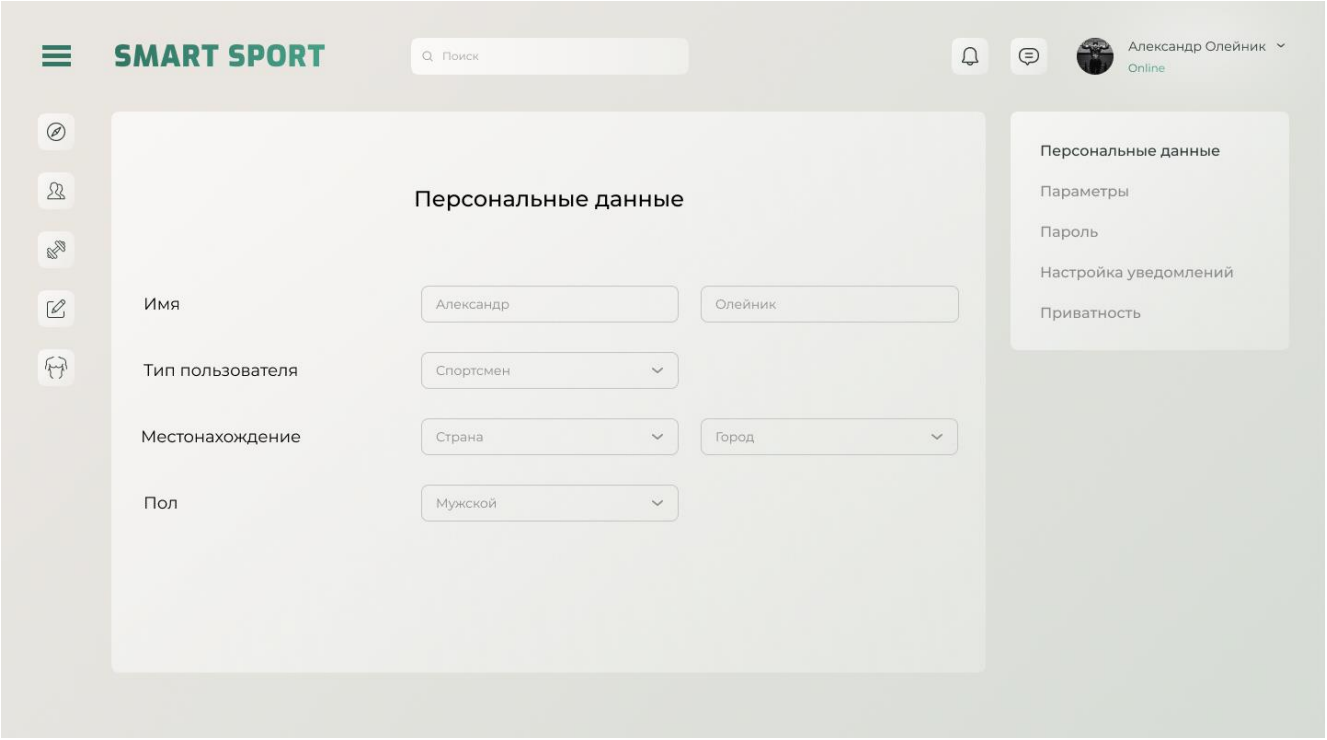



Рисунок 5.4.2 – Сторінка редагування даних профілю

Під час заповнення форми відбувається валідація введених даних з використанням пакету vee-validate, який дозволяє валідувати дані за допомогою великої кількості готових правил та можливістю створювати свої. Після редагування даних та їх перевірки на коректність вони надсилаються за адресою /api/users запитом типу PUT, якщо оновлення пройшло успішно оновлені дані користувача в vuex зберігаються та відображається повідомлення користувачу про успіх або повідомлення про помилку, відповідно.

## 5.5 Профіль користувача

На сторінці користувачів доступною за адресом /users користувач може виконати пошук користувачів за допомогою фільтрів, які знаходяться справа на сторінці. Для зручності управління фільтрами їх значення необхідно продублювати в адресну строку браузера у вигляди query параметрів, це допоможе зберегти значення фільтрів під час перезавантаження сторінки. Після кожної зміни параметрів фільтрів відбувається запит до серверу для отримання списку профілів користувачів, запит та збереження результату реалізовано через vuex.

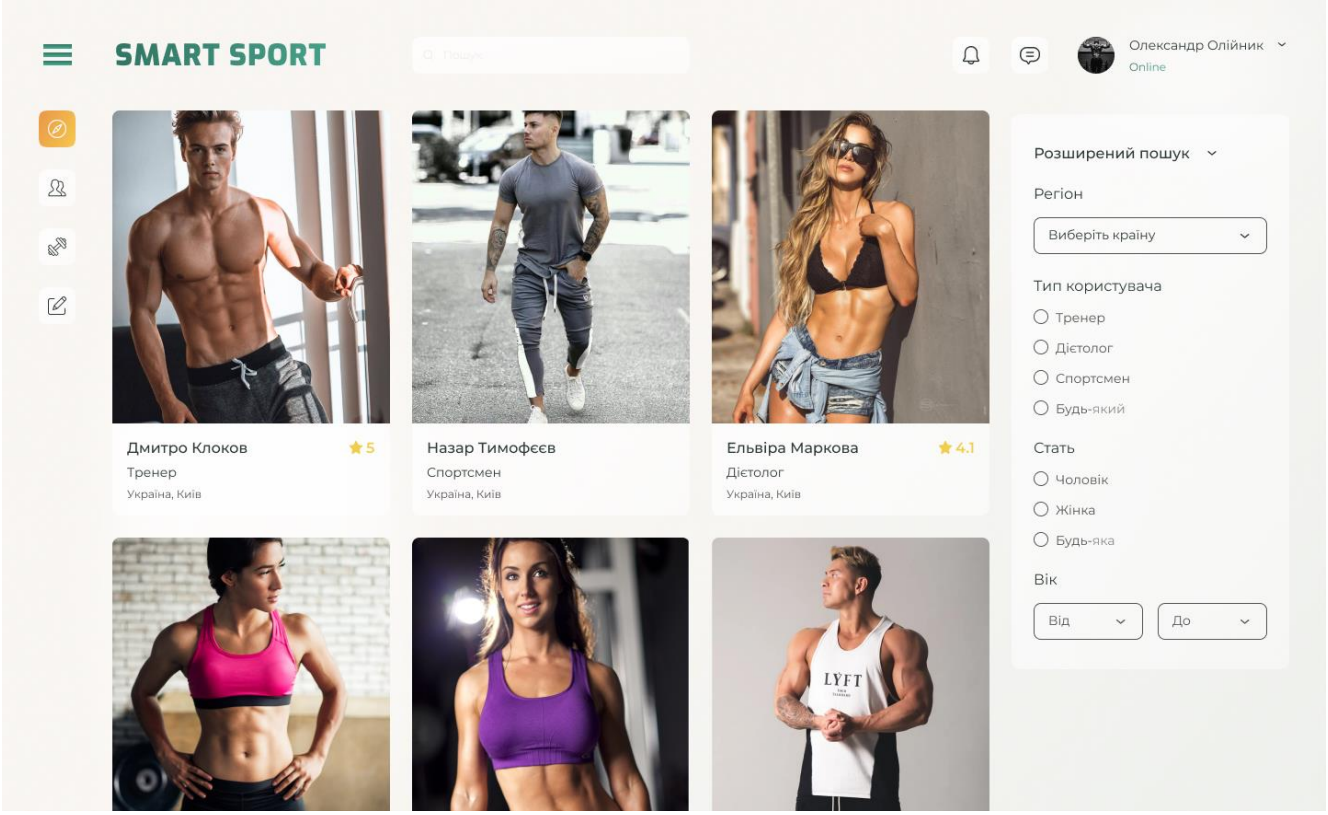



Рисунок 5.4.3 – Сторінка пошуку користувачів

*Підписка та відписка на профілі користувачів.* Користувачі застосунку можуть переглядати профіль один одного, для цього необхідно натиснути на ім’я або зображення користувача, далі відбудеться перехід на адресу /profiles/:id, де :id – унікальний номер користувача. Підчас переходу буде виконано запит типу GET на сервер, щоб отримати дані профілю та відобразити їх. Також на сторінці є кнопка

«Підписатися», якщо користувач не підписаний або «Відписатися» в іншому випадку. Після натискання на кнопку надсилається запит на сервер для підписки або відписки від профілю користувача. У відповіді від сервера отримується оновлений профіль користувача зі зміненими полем following, що відповідає за актуальний стан підписки на профіль.

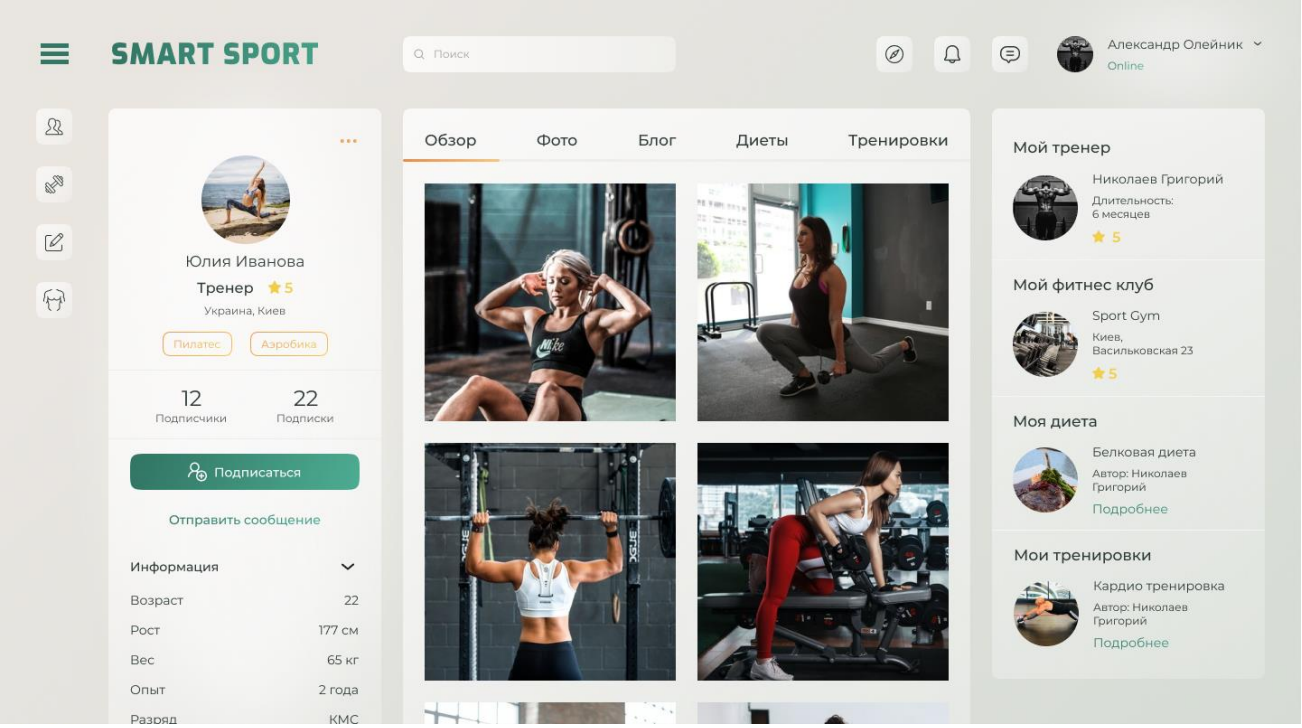



Рисунок 5.4.4 – Сторінка профілю користувачів

*Створення та редагування статті.* Сторінка створення статті розташована за адресом /articles/create, данні з форми знаходяться локально в компоненті. Під час вводу даних в форму відбувається їх валідація за допомогою vee-validate для уникнення проблем під час створення статті на сервері. Форма складається з назви статті, картинки (відображається на карточці статті та на початку статті), короткий опис, список тегів (необхідно вибрати зі списку або створити свої) та основного контенту.

Основний контент статті вноситься з використанням пакету vue2-editor, що дозволяє отримати текст статті у форматі розмітки html. Натиснувши кнопку опублікувати надсилаємо запит на сервер за адресом /api/articles, у разі успішного створення отримуємо об’єкт створеної статті та перенаправляємо користувача до

списку його статей с особистому профілі. Якщо відповідь від сервера містить повідомлення про помилку – виводиться повідомлення користувачу про похибку та залишаємо його на сторінці створення.

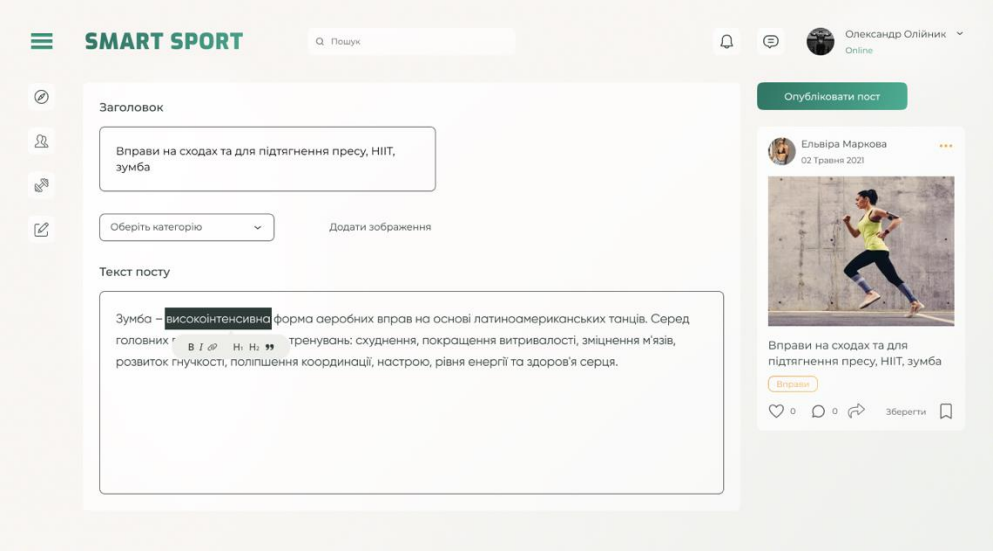



Рисунок 5.4.5 – Створення статті

Для редагування статті необхідно в списку створених статей у особистому кабінеті натиснути на кнопку «Редагувати», що призведе до переходу на сторінку редагування статті.

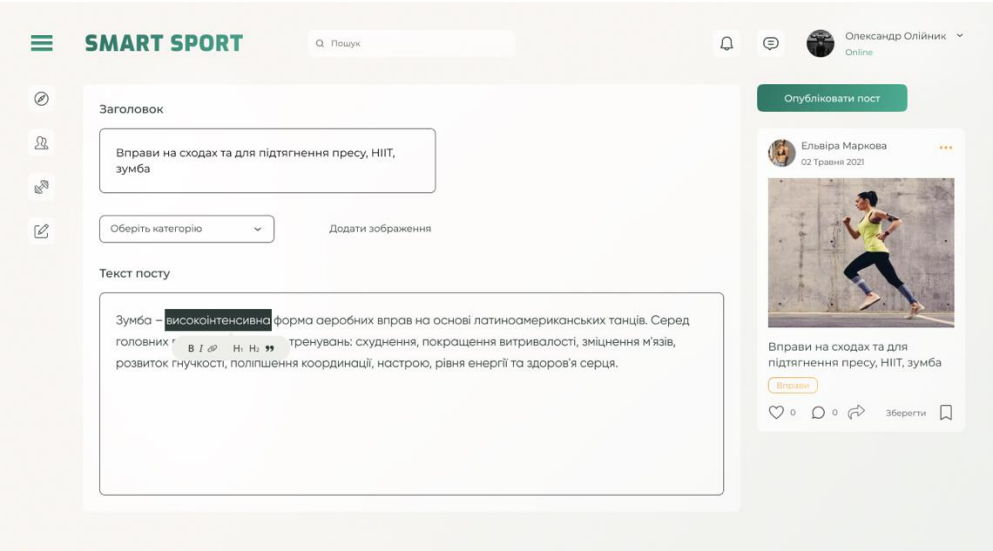



Рисунок 5.4.6 – Редагування статті

Під час переходу з сервера буде отримана стаття для редагування, як тільки вона буде отримана індикатор завантаження закривається та відображається форма редагування, яка є ідентичною до форми створення статті. Редагована стаття буде відправлена до серверу.

*Створення вправ*. Сторінка створення вправ розміщена за адресом /create/exercise, щоб потрапити на дану сторінку необхідно в хедері натиснути на своє ім’я та у відкритому списку дій вибрати «Додати вправу» та бути авторизованим в застосунку та мати тип профілю «Тренер». Якщо користувач не авторизований в його звичайно не буде такого списку дій, проте перейшовши по адресу сторінки його буде переправлено на сторінку авторизації. Заповнивши форму створення вправи та натиснувши на кнопку «Додати» - відправляємо запит до серверу. Після успішного створення відбувається перенаправлення користувача на сторінку створених вправ, під час завантаження сторінки отримується актуальний список вправ.

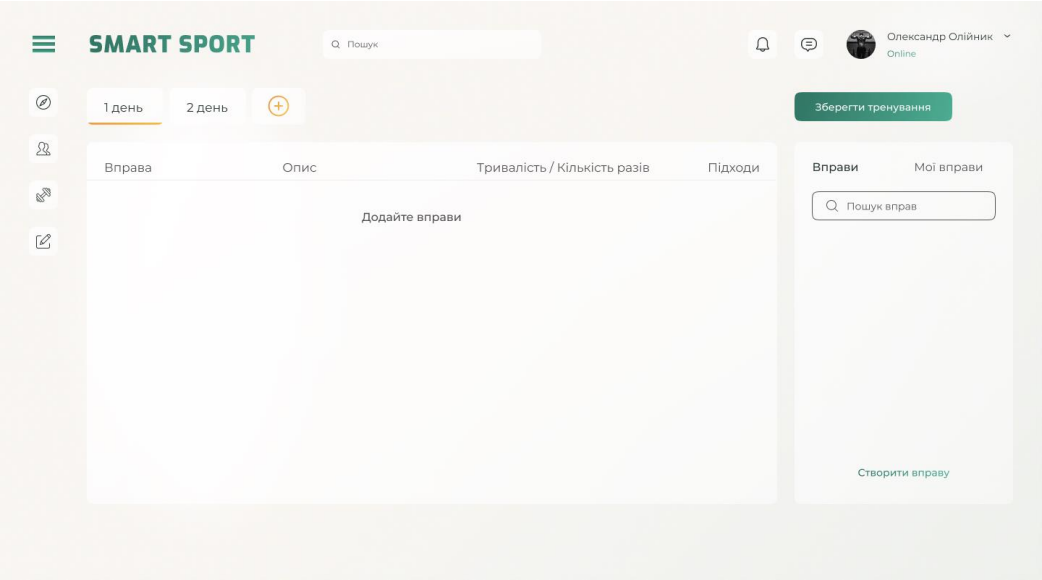



Рисунок 5.4.7 – Створення вправ

*Створення дієт*. Сторінка створення дієти дещо схожа на сторінку створення вправ, проте має деякі відмінності: вибір типу дієти (безкоштовна або платна), та ціна (якщо обрано тип – платна). За допомогою використання редактору vue2-editor можна створювати таблиці для зручного форматування дієти, додавати зображення. Для збереження дієти необхідно натиснути на кнопку «Додати дієту»

після чого буде надіслано запит до серверу. Якщо створення відбудеться успішно відбувається повернення користувача до списку створених дієт.

*Стрічка новин.* Головної сторінкою застосунку для авторизованого користувача є стрічка новин, вона буде містити статті, вправи та дієти користувачів на яких підписаний користувач. Так як стрічка новин може містити багато різної інформації для спрощення пошуку та підбору необхідних матеріалів з правої сторони від списку знаходиться блок фільтрів. Для збереження актуального стану фільтрів та списку відфільтрованих матеріалів всі запити та збереження даних робимо через vuex. Було створено окрему частину vuex під назвою feed, щоб було зручно використовувати його. Для переходу на дану сторінку з будь-якої сторінки застосунку винесено кнопку в хедер (рис 6.4.8 ).

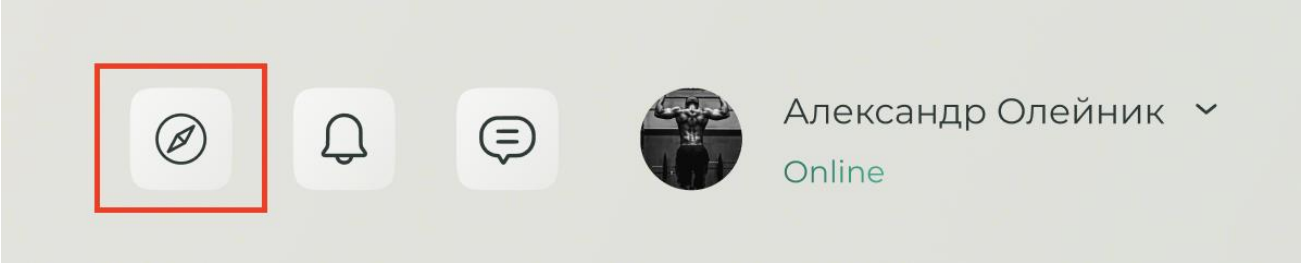



Рисунок 5.4.8 – Хедер застосунку

Безпосередньо перед переходом на сторінку відбувається запит до серверу для отримання актуального списку «новин» за допомогою методу asyncData, який доступний лише для сторінок nuxt. Після отримання індикатор завантаження закривається та відбудеться перехід на сторінку /feed з уже завантаженими даними.

*Перегляд статті.* Після натискання на блок статті, зображення або назву, користувача буде переправлено на сторінку перегляду статті. Данна сторінка складається з декількох блоків:

- ArticleBody - блок статті;
- ArticleAuthor - блок автора;
- ArticleComments - блок коментарів;
- ArticleActions -блок дій.

Блок статті відображає статтю отриману з сервера у наступному порядку: назва, головне зображення, дата створення або редагування, тіло статті. Блок

автора містить зображення профілю, ім’я та прізвище, тип профілю (тренер або спортсмен), кнопку «Підписатися» або «Відписатися». Блок коментарів відображає список коментарів наявних до даної статті за допомогою сервісу disqus та плагіну vue-discus. Блок дій містить кнопки «Вподобати», «Поділитися» та «Прокоментувати».

Висновки до розділу 5

В даному розділі було описано реалізацію клієнтської частини з використанням фреймворку nuxt.js, та додаткових пакетів vue.js, vuex, axios, nuxt-auth, moment. Було визначено основні переваги застосування фреймворку nuxt.js, ддо яких належать: швидкість розробки, використання сучасних технологій, та швидкість роботи клієнтської частини.

# 6 ТЕСТУВАННЯ

Тестування програмного забезпечення – це організаційний процес розробки програмного забезпечення, в рамках якого програмне забезпечення перевіряється на правильність, якість та ефективність. Тестування програмного забезпечення використовується для забезпечення правильної поведінки очікуваних бізнес-систем та продуктів. Тестування програмного забезпечення може бути як ручним, так і автоматизованим процесом.

Ручне тестування програмного забезпечення веде команда або особа, яка буде вручну керувати програмним продуктом та забезпечувати його належну поведінку.

Автоматизоване тестування програмного забезпечення складається з безлічі різних інструментів, які мають різні можливості, починаючи від ізольованих перевірок правильності коду і закінчуючи моделюванням повного досвіду ручного тестування, керованого людиною.

Автоматизоване тестування – це застосування програмних засобів для автоматизації ручного процесу перевірки та перевірки програмного продукту, керованого людиною. Більшість сучасних програмних проектів зараз включають автоматичне тестування з самого початку.

Тоді, коли ручне тестування було нормою, для програмних компаній було звичайною практикою наймати штатну команду з контролю якості. Ця команда повинна розробити збірник «планів тестування» або поетапні контрольні списки, які стверджують, що особливість програмного проекту поводиться належним чином. Потім команда контролю якості виконувала ці контрольні списки вручну кожного разу, коли нове оновлення або зміна надсилалася до програмного проекту, а потім повертала результати планів випробувань інженерній групі для ознайомлення та подальшої розробки для вирішення проблем.

Цей процес був повільним, дорогим та схильним до помилок. Автоматизоване тестування приносить величезні вигоди для підвищення ефективності роботи та рентабельності інвестицій команд із забезпечення якості.

*Наскрізні тести.* Можливо, найціннішими тестами для впровадження є наскрізні (E2E) тести. Тести E2E імітують досвід користувача на рівні всього програмного продукту. Плани тестування E2E, як правило, охоплюють історії на рівні користувача, такі як: «користувач може ввійти в систему», «користувач може внести депозит», «користувач може змінити налаштування електронної пошти». Ці тести надзвичайно цінні для впровадження, оскільки вони гарантують, що реальні користувачі мають певний досвід без помилок, навіть коли натискаються нові коміти.

Засоби тестування E2E фіксують і відтворюють дії користувача, тому плани тестування E2E потім стають записами ключових потоків користувацького досвіду. Якщо програмному продукту бракує будь-якого виду автоматизованого тестування, він отримає найбільшу цінність завдяки впровадженню тестів E2E з найбільш критичних бізнес-потоків. Тести E2E можуть бути дорогими вперед, щоб зафіксувати та записати послідовність потоків користувачів. Якщо програмний продукт не робить швидких щоденних випусків, може бути більш економічним, якщо людська команда виконає вручну плани тестування E2E.

*Unit тести.* Як випливає з назви, модульні тести охоплюють окремі одиниці коду. Одиниці коду найкраще вимірювати у визначеннях функцій. Одиничний тест охоплюватиме окрему функцію. Блокові тести підтвердять, що очікуваний вхід у функцію відповідає очікуваному результату. Кодекс, який має делікатні розрахунки (як, можливо, це стосується фінансів, охорони здоров’я чи аерокосмічної галузі), найкраще охоплювати модульні тести. Блокові випробування є недорогими та швидкими у впровадженні та забезпечують високу рентабельність інвестицій.

*Інтеграційні тести.* Часто одиниця коду здійснює зовнішній дзвінок до сторонньої служби. Первинна кодова база, що перевіряється, не матиме доступу до коду цієї сторонньої утиліти. Тести інтеграції стосуються імітації над цими

сторонніми залежностями та твердження коду, що взаємодіє з ними, поводиться належним чином.

Інтеграційні тести схожі на модульні тести як за їх написанням, так і за їх інструментарієм. Інтеграційні тести можуть бути недорогою альтернативою тестам E2E, однак рентабельність інвестицій є спірною, коли поєднання модульних тестів та E2E вже існує.

*Тести продуктивності.* У контексті розробки програмного забезпечення «продуктивність» використовується для опису швидкості та швидкості реагування програмного проекту. Прикладами показників ефективності є: час завантаження сторінки, час першого відтворення, час відгуку результатів пошуку. Тести продуктивності створюють вимірювання та твердження для цих прикладів. Автоматизовані тести продуктивності запускатимуть тестові випадки за цими показниками, а потім попереджати команду про будь-які регресії або втрату швидкості.

Jest – це тестова система з відкритим кодом, побудована на JavaScript, розроблена в основному для роботи з веб-додатками на основі React та React Native. Часто юніт-тести не надто корисні при запуску на фронтенді будь-якого програмного забезпечення. Це головним чином тому, що модульні тести для інтерфейсу вимагають великої, трудомісткої конфігурації.

Jest можна використовувати для перевірки майже всього що стосується JavaScript, особливо візуалізації браузера веб-додатків. Jest також широко вподобаний для автоматизованого тестування браузера, що робить його однією з найпопулярніших існуючих платформ тестування Javascript.

Крім того, Jest надає змішаний пакет бібліотеки тверджень, разом із тестовим бігуном та вбудованою бібліотекою глузувань. Він виділяється своєю простотою, що робить його ідеальним інструментом для тестування проектів бібліотеки JavaScript, таких як AngularJS, Vue JS, Node JS, Babel та TypeScript.

## 6.1 Тестування серверу

Тестування серверної частини розглянемо на прикладі тестування модулю для роботи з тегами.

Клас Test корисний для забезпечення контексту виконання програми, який по суті висміює повний час виконання Nest, але надає вам хуки, які спрощують управління екземплярами класу, включаючи мокування та заміну. Клас Test має метод createTestingModule (), який приймає за аргумент об’єкт метаданих модуля (той самий об’єкт, який ви передаєте в декоратор @Module ()). Цей метод повертає екземпляр TestingModule, який, у свою чергу, надає декілька методів. Для модульних тестів важливим є метод compile (). Цей метод завантажує модуль із його залежностями (подібно до способу завантаження програми у звичайному файлі main.ts за допомогою NestFactory.create ()) і повертає модуль, готовий до тестування.

У наведеному нижче рисунку (рис) написано тестові приклади для tags.controller.ts, де відображено багато функцій. Розглянемо про кожного, по одному:

describe: слід використовувати для групування подібних одиничних тестових випадків.

beforeEach: Функція Jest, яка запускає код всередині блоку перед кожним тестовим випадком. У нас є інші подібні функції, такі як beforeAll, afterEach і afterAll, і ми повинні використовувати ці функції відповідно до потреби.

createTestingModule: Цей метод створює екземпляр TestingModule, який можна використовувати для отримання будь-яких залежностей, необхідних тестовим кейсам, за допомогою методу «get». У нашому випадку необхідною залежністю є CatsService.

it: Незалежний одиничний тестовий приклад, або ми можемо сказати одноразовий випадок використання, тому слово «опис» може мати кілька «це».

spyOn: spyOn використовується, щоб змусити функцію повернути потрібний результат. Він висміює функцію і повертає результат, який ми хочемо використовувати під час тестування.

expect: Функція очікування використовується для перевірки результату роботи потрібної функції. Отже, у нашому випадку ми шпигуємо за методом findAll TagsService і змушуємо його повернути список тегів, який був доданий раніше. Ми можемо очікувати, що метод TagsController findAll поверне те саме.

```
7  describe('TagController', () => {
8    let tagController: TagController;
9    let tagService: TagService;
10
11   beforeEach(async () => {      Lukas Jakob, 3 years ago • feat: Add tag controller test
12     const module = await Test.createTestingModule({
13       imports: [TypeOrmModule.forRoot(), TypeOrmModule.forFeature([TagEntity])],
14       controllers: [TagController],
15       providers: [TagService],
16     }).compile();
17
18     tagService = module.get<TagService>(TagService);
19     tagController = module.get<TagController>(TagController);
20   });
21
22   describe('findAll', () => {
23     it('should return an array of tags', async () => {
24       const tags : TagEntity[] = [];
25       const createTag = (id, name) => {
26         const tag = new TagEntity();
27         tag.id = id;
28         tag.tag = name;
29         return tag;
30       }
31       tags.push(createTag(1, 'angularjs'));
32       tags.push(createTag(2, 'reactjs'));
33
34       jest.spyOn(tagService, 'findAll').mockImplementation(() => Promise.resolve(tags));
35
36       const findAllResult = await tagController.findAll();
37       expect(findAllResult).toBe(tags);
38     });
39   });
40 });
```

Рисунок 6.1.1 – Фрагмент коду тестування контролеру

## 6.2 Тестування клієнта

Тестування клієнтської частини розглянемо на двох прикладах: тестуванні сторінок nuxt, тестуванні vue компонентів та vuex.

Для тестування сторінок nuxt створюється виконавчий файл тесту з форматом spec.js та імпортуємо до його необідні залежності: константи та конфігурацію nuxt. Перед кожним тестом створюється екземпляр nuxt та запускаємо сервер за адресом вказаному в константах або localhost. Тестування головної сторінки складається з трьох блоків:

1. Тестування головної сторінки на вміст певного тексту;
2. Тестування отримання головної сторінки від серверу, шляхом перевірки коду статусу запиту;
3. Тестування отримання від серверу неіснуючої сторінки також шляхом перевірки коду статусу запиту.

Після проходження всіх тестів знищуємо всі створенні екземпляри nuxt шляхом виклику методу close.

```
let nuxt = null

beforeAll(async () => {
  nuxt = new Nuxt({ ...nuxtConfig, buildDir: constants.buildDir })

  await nuxt.server.listen(constants.port, 'localhost')

}, 300000)


describe('GET /', () => {
  test('Route / exits and render HTML', async () => {
    const { html } = await nuxt.server.renderRoute('/', {})

    expect(html).toContain('Vuetify')
  })
})

describe('GET /', () => {
  test('returns status code 200', async () => {
    const response = await request(nuxt.server.app).get('/')
    expect(response.statusCode).toBe(200)
  })
})

describe('GET /test', () => {
  test('returns status code 404', async () => {
    const response = await request(nuxt.server.app).get('/test')
    expect(response.statusCode).toBe(404)
  })
})


afterAll(() => {
  nuxt.close()
})
```

Рисунок 6.1.2 – Фрагмент коду тестування сторінок

Тестування vuex викоонується за допомогою пакету jest тому, що getters та mutations є чистими функціями і щоб їх протестувати немає потреби в пакеті @vue/test-utils. Кожну частину vuex тестується окремо розбиваючи на дві частини:

1. Тестування getters – перевіряємо чи вірно повертають значення(рис);
2. Тестування mutations – перевіряємо чи вірно дані записуються(рис);

```
import getters from './getters'

test('getArticlesList getters return array of articles from state', () => {
  const state = {
    articles: [
        {
            id: 1,
            title: 'Test article'
        },
        {
            id: 1,
            title: 'Test article'
        },
    ]
  }

  expect(getters.getArticlesList(state).length).toBe(2)
})

test('getArticlesList getters return an empty array of articles from state', () => {
  const state = {
    articles: []
  }
  expect(getters.getArticlesList(state).length).toBe(0)
})
```

Рисунок 6.1.3 – Фрагмент коду тестування гетерів vuex

```
import mutations from './mutations'

test('muttation setArticles set payload to state.articles', () => {
  const state = {
    articles: []
  }

  const payload = [
        {
            id: 1,
            title: 'Test article'
        },
    ]

  mutations.setArticles(state, payload)
  expect(state.articles.length).toBe(1)
})
```

Рисунок 6.1.4 – Фрагмент коду тестування мутацій vuex

Тестування компонентів розглянемо на прикладі тестування компоненту Loader.vue, який відображає індикатор загрузки на екрані. Для цього використовується пакет @vue/test-utils, а саме метод mount для створення об’єкту Wrapper, що буде містити в собі змонтований та відрендерений компонент, його поля та методи доступні для тестування. Спочатку монтується компонент, потім перевіряється, що змонтований компонент не містить дочірніх вузлів тому, що його початковий стан false. Далі перевіряємо чи буде містити компонент дочірній вузол з індикатором загрузки, якщо значення поля відповідального за відображення true.

```
import { mount } from '@vue/test-utils'
import Loader from '@/components/Loader'

describe('Loader.vue', () => {
  const wrapper = mount(Loader)

  it('is spinner removes if show if false', () => {
    expect(wrapper.isEmpty()).toBe(true)
  })


  it('is spinner exist if show true', () => {
    wrapper.vm.show = true
    expect(wrapper.contains('<div class="spinner"></div>')).toBe(true)
  })
})
```

Рисунок 6.1.5 – Фрагмент коду тестування компоненту

## Висновки до розділу 6

Результатом розділу є розглянуто та проаналізовано типів тестування, вибраний тип тестування для використання. Реалізовано та описано тестування основних компонентів застсунку, а саме серверу, компонентів, сховища даних та сторінок клієнту.

# 7 РОЗРОБКА СТАРТАП ПРОЕКТУ

В даному розділі буде представлено аналіз стартап проекту «SmartSport».

## 7.1 Опис ідеї проекту

Опис роздробленого проекту представлено у таблиці 7.1 у вигляді інформаційної карти.

Таблиця 7.1 Інформаційна карта проекту

| | |
|---|---|
| 1. Назва проекту | «SmartSport» |
| 2. Автори проекту | Олійник О.С. |
| 3. Коротка анотація (не більше 1/3 сторінки) | На даний час займатися спортом в фінтес залі або вдома стає дедалі популярнішим. Але для ефективного зайняття спортом в результаті якого можливо отримати очікуванні результати без персонального тренера дуже складно або зовсім неможливо. Тому в залі люди наймають персонального тренера, який буде складати її персональні тренування та правильне харчування, але це коштує не малих грошей. Саме тому, я вирів створити онлайн платформу, яка дає змогу тренерам створювати безкоштовні так і платні програми для користувачів в інтернеті. А користувачі мають можливість знайти будь-яку програму для тренувань, навіть розроблену індивідуально під себе. |
| | 12 місяців |
| | *Тривалість проекту (в місяцях)* |
| 5. Необхідні ресурси | Фінансові:<br>- ~ 4000грн на оренду серверів<br>- ~ 300грн на покупку доменного імені |

| | |
|---|---|
| | Матеріальні:<br>- Доменне ім’я<br>- Комп’ютер або ноутбук<br>Інтелектуальні:<br>- Знання мов програмування<br>- Тестування продукту<br>- Дослідження ринку |
| 6. Опис проблеми, яку вирішує проект | Починаючи заняття спортом люди не знають, які вправи необхідно робити: щоб досягти успіху в цьому, та починають шукати програми тренувань в інтернеті. Однак програми тренувань повинні бути збалансовані та підібрані під кожну людину, є варіант користуватися послугами персонального тренера, але це коштує немалих грошей. |
| 7. Головні цілі та завдання проекту | Ціль: Спростити процес пошуку та підбору програм тренування для людей, які займаються спортом<br>Завдання: Створити зручний застосунок, який буде допомагати знаходити програми тренувань та полегшувати процес зайняття спортом |
| 8. Очікувані результати | |
| Для заняття спортом не потрібно буде платити великі кошти за послуги тренера, пошук програм буде набагато простіший та швидкий, нація буде здоровою. Тренери створивши свої програми тренувань зможуть мати додатковий дохід. | |

Отже, основна проблема, яку вирішує розробка даного стартап-проекту є спрощення процесу пошуку та підбору програм тренування для людей, які займаються спортом.

## 7.2 Технологічний аудит ідеї проекту

MVP (Мінімально життєздатний продукт) – це версія продукту, що дозволяє запустити цикл «створити-оцінити-навчитися» з мінімальними зусиллями, витративши якнайменше часу на розробку [31].

У таблиці 7.2 представлено автоматизований MVP.

Таблиця 7.2 Формування MVP продукту стартапу

| | |
|---|---|
| Проблема, що вирішується | Людству, для того, щоб бути здоровими потрібно займатися спортом, для того щоб заняття були ефективними необхідно дотримуватись певного плану в заняттях та дієти. |
| Ідея продукту | Спрощення процесу пошуку та підбору програм тренування для людей, які займаються спортом |
| MVP 1 | Усне мовлення. Люди передавали свої знання на словах, але це вузьке коло людей. Як люди поза цього кола дізнаються цю інформацію |
| MVP 2 | Професійна література, в якій пояснюється що потрібно робити та як, але необхідно купувати її та читати для того, щоб зробити висновки для себе |
| MVP 3 | Веб-сайти з готовими програмами тренувань, але необхідно самостійно знайти програму яка вам необхідна, але програми повинні складати професіонали та підбирати про вас |
| MVP 4 | Веб-додатки. Допомагають з’єднати тренерів та спортсменів на одній платформі, де зручно можна обмінюватись програмами тренувань та заробляти на цьому кошти |

В таблиці 7.2 було продемонстровано процес автоматизації обробки, починаючи з використання усного мовлення для передачі знань до веб додатків, що допомагають найти тренерів з якими можна обмінюватись програмами тренувань та заробляти на цьому кошти.

## 7.3 Розроблення ринкової та маркетингової стратегії проекту

Розроблення ринкової стратегії передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів, що наведено а таблиці 7.3.

Таблиця 7.3 Вибір цільових груп потенційних споживачів

| № п/п | Опис профілю цільової групи потенційних клієнтів | Готовність споживачів сприйняти продукт | Орієнтовний попит в межах цільової групи (сегменту) | Інтенсивність конкуренції в сегменті | Простота входу у сегмент |
|---|---|---|---|---|---|
| | Фітнес-тренери | Готові, 90% | Високий: можливість отримати додатковій дохід | Низька: не багато платформ для отримання доходу | Просто: не існує конкуренції |
| | Люди які займаються спортом | Готові, 90% | Високий: багато спортсменів не мають правильно зіставлених | Середня: існують веб-сайти з готовими програмами для тренувань | Середня: існують сайти з подібним контентом |

| | | | | | |
|---|---|---|---|---|---|
| | | | прогрес для тренувань | | |
| | Дієтологи | Готові, 60% | Високий: можливість отримати додатковій дохід | Середня: існують веб-сайти з інформацією про дієти | Середня: існують сайти з такою інформацією |
| Які цільові групи обрано: Дієтологи, фітнес-тренери, спортсмени | | | | | |

Для роботи в обраних сегментах ринку необхідно сформувати базову стратегію розвитку, що представлено в таблиці 7.4.

Таблиця 7.4 Визначення базової стратегії розвитку

| № п/п | Обрана альтернатива розвитку проекту | Стратегія охоплення ринку | Ключові конкурентоспроможні позиції відповідно до обраної альтернативи | Базова стратегія розвитку* |
|---|---|---|---|---|
| | Створення збірників програм для тренування | Масовий маркетинг(реклама в соціальних мережах, партнерство з спорт-залами) | Зручність, доступність, вирішення проблем | Стратегія спеціалізації |

Наступним кроком є вибір стратегії конкурентної поведінки, пропозиція якої наведена в таблиці 7.5.

Таблиця 7.5 Визначення базової стратегії конкурентної поведінки

| № п/п | Чи є проект «першопрохідцем» на ринку? | Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів? | Чи буде компанія копіювати основні характеристики товару конкурента, і які? | Стратегія конкурентної поведінки* |
|---|---|---|---|---|
| | Ні | Шукати та забирати існуючих | Так:<br>- Створення власних вправ;<br>- Створення програм для тренування;<br>- Розміщення програм | Заняття конкурентної ніші |

За основу стратегії конкурентної поведінки обрано стратегію заняття конкурентної ніші, що дозволить збільшити та розширити попит запропонованого додатку для створення та розміщення власних вправ, програм тренувань.

Наступним кроком розробляється стратегія позиціонування (табл. 7.6), що в свою чергу полягає у формуванні ринкової позиції (комплексу асоціацій), за яким споживачі мають ідентифікувати торгівельну марку/проект [30].

Таблиця 7.6 Визначення стратегії позиціонування

| Вимоги до товару цільової аудиторії | Базова стратегія розвитку | Ключові конкурентоспроможні позиції власного стартап-проекту | Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових) |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Зручне користування застосунком для обміну програми для тренувань між тренерами та спортсменами | Стратегія спеціалізації - задоволення потреб цільового сегменту | Створення програм тренерами<br>Можливість продавати/купувати програми<br>Створення дієт<br>Доступність<br>Спілкування між тренером та спортсменом | Створення програм тренерами<br>Можливість продавати/купувати програми<br>Створення дієт |

Першим кроком є формування маркетингової концепції товару, який отримає споживач. Для цього у таблиці 7.7 потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Таблиця 7.7 Визначення ключових переваг концепції потенційного товару

| № п/п | Потреба | Вигода, яку пропонує товар | Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити |
|---|---|---|---|
| 1 | Програми тренувань | Готові програми для тренувань | Створення програм тренерами, можливість продавати/купувати програми, контакт тренера і спортсмена |
| 2 | Програми правильного харчування | Готові дієти | Створення дієт дієтологами, можливість продавати/купувати дієти, контакт дієтолога і спортсмена |

Запропонована технічна реалізація системи може задовольнити основні потреби, що можуть бути висунуті споживачами.

Далі розробляється трирівнева маркетингова модель товару, опис трьох рівнів моделі товару наведено в таблиці 7.8.

Таблиця 7.8 Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td colspan="3">Сутність та складові</td></tr>
<tr><td>I. Товар за задумом</td><td colspan="3">Застосунок для обміну програмами тренувань</td></tr>
<tr><td rowspan="4"></td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
<tr><td>1. Створення програм тренувань<br>2. Обмін повідомленями<br>3. Можливість купити або продати програму тренувань<br>4. Створення вправ<br>5. Створення дієт дієтологами</td><td>Нм<br>Нм<br>Нм<br><br>Нм<br>Нм</td><td>Тх<br>Тх<br>Тх<br><br>Тх<br>Тх</td></tr>
<tr><td colspan="2">Якість: тестування застосунку перед впровадженням, зворотній зв’язок</td></tr>
<tr><td colspan="2">Марка: SmartSport</td></tr>
<tr><td rowspan="2"></td><td colspan="2">До продажу Застосунок в якому можливо обмінюватися програмами для тренувань</td></tr>
<tr><td colspan="2">Після продажу Застосунок з підтримкої користувачів на сайті, яка допоможе вирішувати проблеми користувачів</td></tr>
<tr><td colspan="4">За рахунок чого потенційний товар буде захищено від копіювання: інтелектуальна власність, поєдання характеристи та властивостей</td></tr>
</table>

З таблиці 7.8 можна зробити висновок про проведення опису трьох рівнів моделі товару встановлено та описано задум майбутнього застосунку.

Наступним кроком є визначення цінових меж (табл. 7.9), якими необхідно керуватись при встановленні ціни на потенційний товар.

Таблиця 7.9 Визначення меж встановлення ціни

| № п/п | Рівень цін на товари-замінники | Рівень цін на товари-аналоги | Рівень доходів цільової групи споживачів | Верхня та нижня межі встановлення ціни на товар/послугу |
|---|---|---|---|---|
| | 500 грн | 200-250 грн | Низький/Середній( 4000-7000грн) | 10% комісї від продажу |

Далі проведено визначення оптимальної системи збуту, в межах якого приймається рішення (табл. 7.10).

Таблиця 7.10 Формування системи збуту

| № п/п | Специфіка закупівельної поведінки цільових клієнтів | Функції збуту, які має виконувати постачальник товару | Глибина каналу збуту | Оптимальна система збуту |
|---|---|---|---|---|
| | Орієнтація на регулярні оновлення програм тренування | Формування попиту | 1. Розробник<br>2. Тренер<br>3. Користувач | Власні сили |

Останнім кроком розроблення маркетингової програми є розроблення концепції маркетингових комунікацій (табл. 7.11).

Таблиця 7.11. Концепція маркетингових комунікацій

| № п/п | Специфіка поведінки цільових клієнтів | Канали комунікацій, цільових клієнтів | Ключові позиції, обрані для позиціонування | Завдання рекламного повідомлення | Концепція рекламного звернення |
|---|---|---|---|---|---|
| | Пошук програм для тренування | Соціальні мережі | Зручність<br>Актуальність проблеми<br>Надійність | Мотивація клієнтів скористатися застосунком | Зручний пошук програм тренування та дієт |

Висновки до розділу 7

Результатом розділу є розроблений стартап проект, що може бути впровадженим як технічне рішення «SmartSport» в сучасний ринок у сфері.

Визначено основні цілі та завдання проекту, проведено технологічний аудит ідеї проекту, де було продемонстровано процес автоматизації обробки, починаючи з використання усного мовлення для передачі знань до веб додатків, що допомагають найти тренерів з якими можна обмінюватись програмами тренувань.

Було розглянуто потенційні групи клієнтів, до яких належать спортсмени, тренери та дієтологи, виявлено проект не є «першопрохідцем» на ринку та за основу стратегії конкурентної поведінки обрано стратегію заняття конкурентної ніші, в якості базової стратегії розвитку обрано стратегію спеціалізації, визначено ключові переваги концепції потенційного товару, а отже розроблений проект має перспективи впровадження.

Розроблений застосунок в подальшому буде захищений авторським правом на програму.

# ЗАГАЛЬНІ ВИСНОВКИ

Результатом виконання роботи є створений вебзастосунок для онлайн взаємодії тренерів та їх клієнтами, що може бути впровадженим.

Визначено основні цілі та завдання проекту, аналіз рішень конкурентів та використання їх переваг. Проаналізовано технічні можливості для створення даного застосунку та їх використання. Вебзастосунок розроблений по принципу клієнт-серверного архітектурного підходу, де в ролі сервера виступає фреймворк nest.js з додатковими пакетами та nuxt.js з додатковими пакетами в ролі клієнту, дані зберігається в системі управління базами даних mysql.

Описано процеси створення серверу та клієнту з наведення схематичного зображення основних алгоритмів та таблицями з детальним описом класів. Реалізовано та описано тестування окремих частин вебзастосунку з використанням пакетів jest та vue@test-utils.

Розроблено стартап проект вебзастосунку, де проведено технологічний аудит ідеї проекту, розглянуто потенційні групи клієнтів, обрано базову стратегію розвитку.

# ПЕРЕЛІК ПОСИЛАНЬ

1. Ivan Zmerzlyi. Клієнт-серверна архітектура та ролі серверів. [Електронний ресурс] / Ivan Zmerzlyi. – 2017. – Режим доступу до ресурсу: https://medium.com/@IvanZmerzlyi/%D0%BA%D0%BB%D1%96%D1%94%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B0-%D0%B0%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0-%D1%82%D0%B0-%D1%80%D0%BE%D0%BB%D1%96-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D1%96%D0%B2-9893d8048229.

2. Висоцька В. А. Архітектура та проектування комонентних систем [Електронний ресурс] / В. А. Висоцька. – 2016. – Режим доступу до ресурсу: http://www.victana.lviv.ua/knyhy/konspekty-lektsii/133-kros-platformenne-prohramuvannia-ta-khmarni-servisy/568-lektsiia-20-arkhitektura-ta-proektuvannia-komonentnykh-system-2016-r.

3. Мобильное приложение «MasterTrener» [Електронний ресурс] – Режим доступу до ресурсу: https://www.studio-maximus.com/works/apps/mastertrener/.

4. НАЧНИ ТРЕНИРОВКИ УЖЕ СЕЙЧАС [Електронний ресурс] – Режим доступу до ресурсу: https://trener.ua/ru/kiev.

5. ФитЮнион [Електронний ресурс] – Режим доступу до ресурсу: https://fitunion.pro/.

6. nestjs [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/.

7. nestjs.controllers [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/controllers.

8. nestjs.providers [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/providers.

9. nestjs.modules [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/modules.

10. Gonzalez D. SPAs and Server Side Rendering: A Must, or a Maybe? [Електронний ресурс] / Dean Gonzalez – Режим доступу до ресурсу: https://dotcms.com/blog/post/spas-and-server-side-rendering-a-must-or-a-maybe.

11. GUILLAUME B. Client-side vs. Server-side vs. Pre-rendering for Web Apps [Електронний ресурс] / BREUX GUILLAUME – Режим доступу до ресурсу: https://www.toptal.com/front-end/client-side-vs-server-side-pre-rendering.

12. Grzegorz Laszczak. 1. Ivan Zmerzlyi. Клієнт-серверна архітектура та ролі серверів. [Електронний ресурс] / Ivan Zmerzlyi. – 2017. – Режим доступу до ресурсу: https://medium.com/@IvanZmerzlyi/%D0%BA%D0%BB%D1%96%D1%94%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B [Електронний ресурс] / Grzegorz Laszczak – Режим доступу до ресурсу: https://selleo.com/blog/why-choose-nest-js-as-your-backend-framework.

13. Gabriel Tanner. Nestjs crash course [Електронний ресурс] / Gabriel Tanner. – 2019. – Режим доступу до ресурсу: https://gabrieltanner.org/blog/nestjs-crashcourse.

14. TypeScript - Overview [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.tutorialspoint.com/typescript/typescript_overview.htm.

15. node.bcrypt.js [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/kelektiv/node.bcrypt.js#readme.

16. TypeORM - Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/typeorm/TypeORM%20-%20Introduction.htm#:~:text=TypeORM%20is%20an%20Object%20Relational,js%20and%20written%20in%20TypeScript.&text=js.

17. MySQL - Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/mysql/mysql-introduction.htm.

18. HTTP - Overview [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/http/http_overview.htm.

19. What is Swagger and Why Does it Matter? [Електронний ресурс]. – 10. – Режим доступу до ресурсу: https://blog.readme.com/what-is-swagger-and-why-it-matters/.

20. The Good and the Bad of Vue.js Framework Programming Share:Share on FacebookTweet about this on TwitterShare on LinkedInComment:0 [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.altexsoft.com/blog/engineering/pros-and-cons-of-vue-js/.

21. What is a REST API? [Електронний ресурс] – Режим доступу до ресурсу: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

22. Введение в REST API — RESTful веб-сервисы [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: Введение в REST API — RESTful веб-сервисы.

23. Understanding And Using REST APIs [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/.

24. Introduction to JSON Web Tokens [Електронний ресурс] – Режим доступу до ресурсу: https://jwt.io/introduction.

25. Flavio Copes. JWT authentication: When and how to use it [Електронний ресурс] / Flavio Copes. – 2018. – Режим доступу до ресурсу: https://blog.logrocket.com/jwt-authentication-best-practices/.

26. Building and validating Vue forms the painless way [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://medium.com/js-dojo/building-and-validating-vue-forms-the-painless-way-1624e1399732.

27. The Good and the Bad of Vue.js Framework Programming [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.altexsoft.com/blog/engineering/pros-and-cons-of-vue-js/.

28. The Complete Guide to Building a Full-Blown Multilanguage Website with Nuxt.js [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.storyblok.com/tp/nuxt-js-multilanguage-website-tutorial.

29. Kenneth Brenes. How to Build a Real-world App: A Nuxt.js Tutorial (Part 1) [Електронний ресурс] / Kenneth Brenes. – 2021. – Режим доступу до ресурсу: The Complete Guide to Building a Full-Blown Multilanguage Website with Nuxt.js [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.storyblok.com/tp/nuxt-js-multilanguage-website-tutorial..

30. Getting Started With Nuxt [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: Kenneth Brenes. How to Build a Real-world App: A Nuxt.js Tutorial (Part 1) [Електронний ресурс] / Kenneth Brenes. – 2021. – Режим доступу до ресурсу: https://www.smashingmagazine.com/2020/04/getting-started-nuxt/

31. Axios tutorial [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://zetcode.com/javascript/axios/.

32. Michael Auderer. How To Get Started with Node.js and Express Node.js [Електронний ресурс] / Michael Auderer. – 2021. – Режим доступу до ресурсу: https://zetcode.com/javascript/axios/..

33. Quentin Somerville. Getting started with NestJS & TypeORM (+bonus NestJS Admin) [Електронний ресурс] / Quentin Somerville. – 2019. – Режим доступу до ресурсу: https://blog.theodo.com/2019/05/an-overview-of-nestjs-typeorm-release-your-first-application-in-less-than-30-minutes/.

34. The Good and the Bad of Vue.js Framework Programming Share:Share on FacebookTweet about this on TwitterShare on LinkedInComment:0 [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.altexsoft.com/blog/engineering/pros-and-cons-of-vue-js/.

Додаток А

```
import { NestFactory } from '@nestjs/core';
import { ApplicationModule } from './app.module';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import {PhotosService} from "./photos/photos.service";

async function bootstrap() {
  const appOptions = {cors: true};
  const app = await NestFactory.create(ApplicationModule, appOptions);
  app.setGlobalPrefix('api');

  const options = new DocumentBuilder()
    .setTitle('Fitness api')
    .setDescription('The best api')
    .setVersion('1.0')
    .setBasePath('api')
    .addBearerAuth()
    .build();
  const document = SwaggerModule.createDocument(app, options);
  SwaggerModule.setup('/docs', app, document);

  await app.listen(3000);
  app.get(PhotosService).startWorker();
}
bootstrap();

import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { ArticleModule } from './article/article.module';
import { ExerciseModule } from "./exercise/exercise.module";
import { UserModule } from './user/user.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Connection } from 'typeorm';
import { ProfileModule } from './profile/profile.module';
import { TagModule } from './tag/tag.module';
import {PhotosModule} from "./photos/photos.module";

@Module({
  imports: [
```

```
    TypeOrmModule.forRoot(),
    ArticleModule,
    UserModule,
    ProfileModule,
    TagModule,
    PhotosModule,
    ExerciseModule,
  ],
  controllers: [
    AppController
  ],
  providers: []
})
export class ApplicationModule {
  constructor(private readonly connection: Connection) {}
}

import { Get, Controller } from '@nestjs/common';

@Controller('api')
export class AppController {
  @Get()
  private root(): string {
    return 'Hello! This is fitness API :)';
  }
}

import {Get, Post, Body, Put, Delete, Query, Param, Controller} from
'@nestjs/common';
import { Request } from 'express';
import { ArticleService } from './article.service';
import {AddExercise, CreateArticleDto, CreateCommentDto} from './dto';
import { ArticlesRO, ArticleRO } from './article.interface';
import { CommentsRO } from './article.interface';
import { User } from '../user/user.decorator';

import {
  ApiUseTags,
  ApiBearerAuth,
  ApiResponse,
```

```
  ApiOperation,
} from '@nestjs/swagger';
import {ExerciseRO, ExercisesRO} from "../exercise/exercise.interface";

@ApiBearerAuth()
@ApiUseTags('articles')
@Controller('articles')
export class ArticleController {

  constructor(private readonly articleService: ArticleService) {}

  @ApiOperation({ title: 'Get all articles' })
  @ApiResponse({ status: 200, description: 'Return all articles.'})
  @Get()
  async findAll(@Query() query): Promise<ArticlesRO> {
    return await this.articleService.findAll(query);
  }

  @Get(':slug')
  async findOne(@Param('slug') slug): Promise<ArticleRO> {
    return await this.articleService.findOne({slug});
  }

  @Get(':slug/comments')
  async findComments(@Param('slug') slug): Promise<CommentsRO> {
    return await this.articleService.findComments(slug);
  }
  @Get(':slug/exercises')
  async findExercises(@Param('slug') slug): Promise<ExercisesRO> {
    return await this.articleService.findExercises(slug);
  }

  @ApiOperation({ title: 'Create article' })
  @ApiResponse({ status: 201, description: 'The article has been successfully created.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Post()
  async create(@Body('id') userId: number, @Body('article') articleData:
CreateArticleDto) {
    return this.articleService.create(userId, articleData);
  }
```

```
  @ApiOperation({ title: 'Update article' })
  @ApiResponse({ status: 201, description: 'The article has been successfully
updated.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Put(':slug')
  async update(@Param() params, @Body('article') articleData: CreateArticleDto) {
    // Todo: update slug also when title gets changed
    return this.articleService.update(params.slug, articleData);
  }

  @ApiOperation({ title: 'Delete article' })
  @ApiResponse({ status: 201, description: 'The article has been successfully deleted.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Delete(':slug')
  async delete(@Param() params) {
    return this.articleService.delete(params.slug);
  }

  @ApiOperation({ title: 'Create comment' })
  @ApiResponse({ status: 201, description: 'The comment has been successfully
created.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Post(':slug/comments')
  async createComment(@Param('slug') slug, @Body('comment') commentData:
CreateCommentDto) {
    return await this.articleService.addComment(slug, commentData);
  }

  @ApiOperation({ title: 'Delete comment' })
  @ApiResponse({ status: 201, description: 'The article has been successfully deleted.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Delete(':slug/comments/:id')
  async deleteComment(@Param() params) {
    const {slug, id} = params;
    return await this.articleService.deleteComment(slug, id);
  }

  @ApiOperation({ title: 'Favorite article' })
```

```
  @ApiResponse({ status: 201, description: 'The article has been successfully
favorited.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Post(':slug/favorite')
  async favorite(@User('id') userId: number, @Param('slug') slug) {
    return await this.articleService.favorite(userId, slug);
  }

  @ApiOperation({ title: 'Unfavorite article' })
  @ApiResponse({ status: 201, description: 'The article has been successfully
unfavorited.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Delete(':slug/favorite')
  async unFavorite(@User('id') userId: number, @Param('slug') slug) {
    return await this.articleService.unFavorite(userId, slug);
  }

  @ApiOperation({ title: 'Get article feed' })
  @ApiResponse({ status: 200, description: 'Return article feed.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Get('feed')
  async getFeed(@User('id') userId: number, @Query() query): Promise<ArticlesRO> {
    return await this.articleService.findFeed(userId, query);
  }

  @ApiOperation({ title: 'Add exercise' })
  @ApiResponse({ status: 201, description: 'The exercise has been successfully
added.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Post(':slug/exercises')
  async addExercise(@Param('slug') slug, @Body() exerciseData: AddExercise) {
    return await this.articleService.addExercise(slug, exerciseData);
  }



  @ApiOperation({ title: 'Delete exercise' })
  @ApiResponse({ status: 201, description: 'The exercise has been successfully
deleted.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
```

```
  @Delete(':slug/exercises/:id')
  async deleteExercise(@Param() params) {
    const {slug, id} = params;
    return await this.articleService.deleteExercise(slug, id);
  }

}

import {
  Entity,
  PrimaryGeneratedColumn,
  Column,
  OneToOne,
  ManyToOne,
  OneToMany,
  JoinColumn,
  AfterUpdate,
  BeforeUpdate,
  ManyToMany, JoinTable
} from 'typeorm';
import { UserEntity } from '../user/user.entity';
import { Comment } from './comment.entity';
import {ExerciseEntity} from "../exercise/exercise.entity";

@Entity('article')
export class ArticleEntity {

  @PrimaryGeneratedColumn()
  private id: number;

  @Column()
  slug: string;

  @Column()
  title: string;

  @Column({default: ''})
  description: string;

  @Column({default: ''})
```

```
  private body: string;

  @Column({ type: 'timestamp', default: () => "CURRENT_TIMESTAMP"})
  private created: Date;

  @Column({ type: 'timestamp', default: () => "CURRENT_TIMESTAMP"})
  private updated: Date;

  @BeforeUpdate()
  private updateTimestamp() {
    this.updated = new Date;
  }

  @Column('simple-array')
  tagList: string[];

  @ManyToOne(type => UserEntity, user => user.articles)
  private author: UserEntity;

  @ManyToMany(type => ExerciseEntity, { eager: true })
  @JoinTable()
  private exercises: ExerciseEntity[];

  @OneToMany(type => Comment, comment => comment.article, {eager: true})
  @JoinColumn()
  comments: Comment[];

  @Column({default: 0})
  private favoriteCount: number;
}

import { UserData } from '../user/user.interface';
import { ArticleEntity } from './article.entity';
interface Comment {
  body: string;
}

interface ArticleData {
  slug: string;
  title: string;
```

```
  description: string;
  body?: string;
  tagList?: string[];
  createdAt?: Date
  updatedAt?: Date
  favorited?: boolean;
  favoritesCount?: number;
  author?: UserData;
}

export interface CommentsRO {
  comments: Comment[];
}

export interface ArticleRO {
  article: ArticleEntity;
}

export interface ArticlesRO {
  articles: ArticleEntity[];
  articlesCount: number;
}



import { MiddlewareConsumer, Module, NestModule, RequestMethod } from
'@nestjs/common';
import { ArticleController } from './article.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ArticleEntity } from './article.entity';
import {ExerciseEntity} from "../exercise/exercise.entity";
import { Comment } from './comment.entity';
import { UserEntity } from '../user/user.entity';
import { FollowsEntity } from '../profile/follows.entity';
import { ArticleService } from './article.service';
import { AuthMiddleware } from '../user/auth.middleware';
import { UserModule } from '../user/user.module';

@Module({
```

```
  imports: [TypeOrmModule.forFeature([ArticleEntity,ExerciseEntity, Comment,
UserEntity, FollowsEntity]), UserModule],
  providers: [ArticleService],
  controllers: [
    ArticleController
  ]
})
export class ArticleModule implements NestModule {
  public configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(AuthMiddleware)
      .forRoutes(
        {path: 'articles/feed', method: RequestMethod.GET},
        {path: 'articles', method: RequestMethod.POST},
        {path: 'articles/:slug', method: RequestMethod.DELETE},
        {path: 'articles/:slug', method: RequestMethod.PUT},
        {path: 'articles/:slug/comments', method: RequestMethod.POST},
        {path: 'articles/:slug/comments/:id', method: RequestMethod.DELETE},
        {path: 'articles/:slug/favorite', method: RequestMethod.POST},
        {path: 'articles/:slug/favorite', method: RequestMethod.DELETE});
  }
}


import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository, getRepository, DeleteResult } from 'typeorm';
import { ArticleEntity } from './article.entity';
import { Comment } from './comment.entity';
import {ExerciseEntity} from "../exercise/exercise.entity";
import { UserEntity } from '../user/user.entity';
import { FollowsEntity } from '../profile/follows.entity';
import { CreateArticleDto } from './dto';

import {ArticleRO, ArticlesRO, CommentsRO} from './article.interface';
import {ExerciseRO, ExercisesRO} from "../exercise/exercise.interface";
const slug = require('slug');

@Injectable()
export class ArticleService {
```

```
constructor(
  @InjectRepository(ArticleEntity)
  private readonly articleRepository: Repository<ArticleEntity>,
  @InjectRepository(ExerciseEntity)
  private readonly exerciseRepository: Repository<ExerciseEntity>,
  @InjectRepository(Comment)
  private readonly commentRepository: Repository<Comment>,
  @InjectRepository(UserEntity)
  private readonly userRepository: Repository<UserEntity>,
  @InjectRepository(FollowsEntity)
  private readonly followsRepository: Repository<FollowsEntity>
) {}

async findAll(query): Promise<ArticlesRO> {

  const qb = await getRepository(ArticleEntity)
    .createQueryBuilder('article')
    .leftJoinAndSelect('article.author', 'author');

  qb.where("1 = 1");

  if ('tag' in query) {
    qb.andWhere("article.tagList LIKE :tag", { tag: `%${query.tag}%` });
  }

  if ('author' in query) {
    const author = await this.userRepository.findOne({username: query.author});
    qb.andWhere("article.authorId = :id", { id: author.id });
  }

  if ('favorited' in query) {
    const author = await this.userRepository.findOne({username: query.favorited});
    const ids = author.favorites.map(el => el.id);
    qb.andWhere("article.authorId IN (:ids)", { ids });
  }

  qb.orderBy('article.created', 'DESC');

  const articlesCount = await qb.getCount();
```

```
  if ('limit' in query) {
    qb.limit(query.limit);
  }

  if ('offset' in query) {
    qb.offset(query.offset);
  }

  const articles = await qb.getMany();

  return {articles, articlesCount};
}

async findFeed(userId: number, query): Promise<ArticlesRO> {
  const _follows = await this.followsRepository.find( {followerId: userId});
  const ids = _follows.map(el => el.followingId);

  const qb = await getRepository(ArticleEntity)
    .createQueryBuilder('article')
    .where('article.authorId IN (:ids)', { ids });

  qb.orderBy('article.created', 'DESC');

  const articlesCount = await qb.getCount();

  if ('limit' in query) {
    qb.limit(query.limit);
  }

  if ('offset' in query) {
    qb.offset(query.offset);
  }

  const articles = await qb.getMany();

  return {articles, articlesCount};
}

async findOne(where): Promise<ArticleRO> {
  const article = await this.articleRepository.findOne(where);
```

```
    return {article};
  }

  async addExercise(slug:string, {id}): Promise<ArticleRO> {
    let article = await this.articleRepository.findOne({slug});
    let exercise = await this.exerciseRepository.findOne({id});
    console.log(article);
    if (!article.exercises) {
      article.exercises = [];
    }
    article.exercises.push(exercise);

    article = await this.articleRepository.save(article);
    return {article}
  }

  async addComment(slug: string, commentData): Promise<ArticleRO> {
    let article = await this.articleRepository.findOne({slug});

    const comment = new Comment();
    comment.body = commentData.body;

    article.comments.push(comment);

    await this.commentRepository.save(comment);
    article = await this.articleRepository.save(article);
    return {article}
  }

  async deleteComment(slug: string, id: string): Promise<ArticleRO> {
    let article = await this.articleRepository.findOne({slug});

    const comment = await this.commentRepository.findOne(id);
    const deleteIndex = article.comments.findIndex(_comment => _comment.id ===
comment.id);

    if (deleteIndex >= 0) {
      const deleteComments = article.comments.splice(deleteIndex, 1);
      await this.commentRepository.delete(deleteComments[0].id);
      article =  await this.articleRepository.save(article);
```

```
      return {article};
    } else {
      return {article};
    }

  }
  async deleteExercise(slug: string, id: string): Promise<ArticleRO> {
    let article = await this.articleRepository.findOne({slug});

    const exercise = await this.exerciseRepository.findOne(id);
    const deleteIndex = article.exercises.findIndex(_exercise => _exercise.id ===
exercise.id);

    if (deleteIndex >= 0) {
      const deleteExercises = article.exercises.splice(deleteIndex, 1);
      await this.exerciseRepository.delete(deleteExercises[0].id);
      article =  await this.articleRepository.save(article);
      return {article};
    } else {
      return {article};
    }

  }


  async favorite(id: number, slug: string): Promise<ArticleRO> {
    let article = await this.articleRepository.findOne({slug});
    const user = await this.userRepository.findOne(id);

    const isNewFavorite = user.favorites.findIndex(_article => _article.id === article.id)
< 0;
    if (isNewFavorite) {
      user.favorites.push(article);
      article.favoriteCount++;

      await this.userRepository.save(user);
      article = await this.articleRepository.save(article);
    }

    return {article};
```

```
}

async unFavorite(id: number, slug: string): Promise<ArticleRO> {
  let article = await this.articleRepository.findOne({slug});
  const user = await this.userRepository.findOne(id);

  const deleteIndex = user.favorites.findIndex(_article => _article.id === article.id);

  if (deleteIndex >= 0) {

    user.favorites.splice(deleteIndex, 1);
    article.favoriteCount--;

    await this.userRepository.save(user);
    article = await this.articleRepository.save(article);
  }

  return {article};
}

async findComments(slug: string): Promise<CommentsRO> {
  const article = await this.articleRepository.findOne({slug});
  return {comments: article.comments};
}

async findExercises(slug: string): Promise<ExercisesRO> {
  const article = await this.articleRepository.findOne({slug});
  return {exercises: article.exercises,exercisesCount: article.exercises.length};
}


async create(userId: number, articleData: CreateArticleDto): Promise<ArticleEntity> {

  let article = new ArticleEntity();
  article.title = articleData.title;
  article.description = articleData.description;
  article.slug = this.slugify(articleData.title);
  article.tagList = articleData.tagList || [];
  article.comments = [];
```

```
    const newArticle = await this.articleRepository.save(article);

    const author = await this.userRepository.findOne({ where: { id: userId } });

    if (Array.isArray(author.articles)) {
      author.articles.push(article);
    } else {
      author.articles = [article];
    }

    await this.userRepository.save(author);

    return newArticle;

  }

  async update(slug: string, articleData: any): Promise<ArticleRO> {
    let toUpdate = await this.articleRepository.findOne({ slug: slug});
    let updated = Object.assign(toUpdate, articleData);
    const article = await this.articleRepository.save(updated);
    return {article};
  }

  async delete(slug: string): Promise<DeleteResult> {
    return await this.articleRepository.delete({ slug: slug});
  }

  private slugify(title: string) {
    return slug(title, {lower: true}) + '-' + (Math.random() * Math.pow(36, 6) |
0).toString(36)
  }
}


import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from 'typeorm';
import { ArticleEntity } from './article.entity';

@Entity()
export class Comment {
```

```
  @PrimaryGeneratedColumn()
  private id: number;

  @Column()
  body: string;

  @ManyToOne(type => ArticleEntity, article => article.comments)
  private article: ArticleEntity;
}

import {Get, Post, Body, Put, Delete, Query, Param, Controller, UsePipes,
ValidationPipe} from '@nestjs/common';
import { Request } from 'express';
import { ExerciseService } from './exercise.service';
import { CreateExerciseDto } from './dto';
import { ExercisesRO, ExerciseRO } from './exercise.interface';
import { User } from '../user/user.decorator';

import {
  ApiUseTags,
  ApiBearerAuth,
  ApiResponse,
  ApiOperation,
} from '@nestjs/swagger';

@ApiBearerAuth()
@ApiUseTags('exercises')
@Controller('exercises')
export class ExerciseController {

  constructor(private readonly exerciseService: ExerciseService) {}

  @ApiOperation({ title: 'Get all exercises' })
  @ApiResponse({ status: 200, description: 'Return all exercises.'})
  @Get()
  async findAll(@Query() query): Promise<ExercisesRO> {
    return await this.exerciseService.findAll(query);
  }
  @ApiOperation({title:'Get exercise by slug'})
  @Get(':slug')
```

```
  async findOne(@Param('slug') slug): Promise<ExerciseRO> {
    return await this.exerciseService.findOne({slug});
  }
  @UsePipes(new ValidationPipe())
  @ApiOperation({ title: 'Create exercise' })
  @ApiResponse({ status: 201, description: 'The exercise has been successfully
created.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Post()
  async create(@Body('id') userId: number, @Body('exercise') exerciseData:
CreateExerciseDto) {
    console.log('dgsdgsdg');
    return this.exerciseService.create(userId, exerciseData);
  }

  @ApiOperation({ title: 'Update exercise' })
  @ApiResponse({ status: 201, description: 'The exercise has been successfully
updated.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Put(':slug')
  async update(@Param() params, @Body('exercise') exerciseData: CreateExerciseDto)
{
    // Todo: update slug also when title gets changed
    return this.exerciseService.update(params.slug, exerciseData);
  }

  @ApiOperation({ title: 'Delete exercise' })
  @ApiResponse({ status: 201, description: 'The exercise has been successfully
deleted.'})
  @ApiResponse({ status: 403, description: 'Forbidden.' })
  @Delete(':slug')
  async delete(@Param() params) {
    return this.exerciseService.delete(params.slug);
  }



}

import { Entity, PrimaryGeneratedColumn, Column,JoinTable, OneToOne,
```

```
ManyToOne,ManyToMany, OneToMany, JoinColumn, AfterUpdate, BeforeUpdate }
from 'typeorm';
import { UserEntity } from '../user/user.entity';
import { ArticleEntity } from '../article/article.entity';

@Entity('exercise')
export class ExerciseEntity {

  @PrimaryGeneratedColumn()
  private id: number;

  @Column({ nullable:true})
  private slug:string;

  @Column()
  title: string;

  @Column({default: ''})
  description: string;

  @Column({default: ''})
  private image: string;

  @Column({ type: 'timestamp', default: () => "CURRENT_TIMESTAMP"})
  private created: Date;

  @Column({ type: 'timestamp', default: () => "CURRENT_TIMESTAMP"})
  private updated: Date;

  @BeforeUpdate()
  private updateTimestamp() {
    this.updated = new Date;
  }

  @Column('simple-array')
  tagList: string[];

  @ManyToOne(type => UserEntity, user => user.exercises)
  private author: UserEntity;
```

```
}

import { UserData } from '../user/user.interface';
import { ExerciseEntity } from './exercise.entity';


interface ExerciseData {
  slug?:string
  title: string;
  description: string;
  image?: string;
  tagList?: string[];
  createdAt?: Date
  updatedAt?: Date
  author?: UserData;
}


export interface ExerciseRO {
  exercise: ExerciseEntity;
}

export interface ExercisesRO {
  exercises: ExerciseEntity[];
  exercisesCount: number;
}



import { MiddlewareConsumer, Module, NestModule, RequestMethod } from
'@nestjs/common';
import { ExerciseController } from './exercise.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ExerciseEntity } from './exercise.entity';
import { UserEntity } from '../user/user.entity';
import { FollowsEntity } from '../profile/follows.entity';
import { ExerciseService } from './exercise.service';
import { AuthMiddleware } from '../user/auth.middleware';
import { UserModule } from '../user/user.module';

@Module({
```

```
  imports: [TypeOrmModule.forFeature([ExerciseEntity, UserEntity, FollowsEntity]),
UserModule],
  providers: [ExerciseService],
  controllers: [
    ExerciseController
  ]
})
export class ExerciseModule implements NestModule {
  public configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(AuthMiddleware)
      .forRoutes(
        {path: 'exercises/feed', method: RequestMethod.GET},
        {path: 'exercises', method: RequestMethod.POST},
        {path: 'exercises/:slug', method: RequestMethod.DELETE},
        {path: 'exercises/:slug', method: RequestMethod.PUT},);
  }
}


import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository, getRepository, DeleteResult } from 'typeorm';
import { ExerciseEntity } from './exercise.entity';
import { UserEntity } from '../user/user.entity';
import { FollowsEntity } from '../profile/follows.entity';
import { CreateExerciseDto } from './dto';

import {ExerciseRO, ExercisesRO} from './exercise.interface';
const slug = require('slug');

@Injectable()
export class ExerciseService {
  constructor(
    @InjectRepository(ExerciseEntity)
    private readonly exerciseRepository: Repository<ExerciseEntity>,
    @InjectRepository(UserEntity)
    private readonly userRepository: Repository<UserEntity>,
    @InjectRepository(FollowsEntity)
    private readonly followsRepository: Repository<FollowsEntity>
```

```
) {}

async findAll(query): Promise<ExercisesRO> {

  const qb = await getRepository(ExerciseEntity)
    .createQueryBuilder('exercise')
    .leftJoinAndSelect('exercise.author', 'author');

  qb.where("1 = 1");

  if ('tag' in query) {
    qb.andWhere("exercise.tagList LIKE :tag", { tag: `%${query.tag}%` });
  }

  if ('author' in query) {
    const author = await this.userRepository.findOne({username: query.author});
    qb.andWhere("exercise.authorId = :id", { id: author.id });
  }


  qb.orderBy('exercise.created', 'DESC');

  const exercisesCount = await qb.getCount();

  if ('limit' in query) {
    qb.limit(query.limit);
  }

  if ('offset' in query) {
    qb.offset(query.offset);
  }

  const exercises = await qb.getMany();

  return {exercises, exercisesCount};
}

async findFeed(userId: number, query): Promise<ExercisesRO> {
  const _follows = await this.followsRepository.find( {followerId: userId});
  const ids = _follows.map(el => el.followingId);
```

```
    const qb = await getRepository(ExerciseEntity)
      .createQueryBuilder('exercise')
      .where('exercise.authorId IN (:ids)', { ids });

    qb.orderBy('exercise.created', 'DESC');

    const exercisesCount = await qb.getCount();

    if ('limit' in query) {
      qb.limit(query.limit);
    }

    if ('offset' in query) {
      qb.offset(query.offset);
    }

    const exercises = await qb.getMany();

    return {exercises, exercisesCount};
  }

  async findOne(where): Promise<ExerciseRO> {
    const exercise = await this.exerciseRepository.findOne(where);
    return {exercise};
  }


  async create(userId: number, exerciseData: CreateExerciseDto):
Promise<ExerciseEntity> {

    let exercise = new ExerciseEntity();
    exercise.title = exerciseData.title;
    exercise.description = exerciseData.description;
    exercise.tagList = exerciseData.tagList || [];


    const newExercise = await this.exerciseRepository.save(exercise);

    const author = await this.userRepository.findOne({ where: { id: userId } });
```

```
    if (Array.isArray(author.exercises)) {
      author.exercises.push(exercise);
    } else {
      author.exercises = [exercise];
    }

    await this.userRepository.save(author);

    return newExercise;

  }

  async update(slug: string, exerciseData: any): Promise<ExerciseRO> {
    let toUpdate = await this.exerciseRepository.findOne({ slug: slug});
    let updated = Object.assign(toUpdate, exerciseData);
    const exercise = await this.exerciseRepository.save(updated);
    return {exercise};
  }

  async delete(slug: string): Promise<DeleteResult> {
    return await this.exerciseRepository.delete({ slug: slug});
  }

  private slugify(title: string) {
    return slug(title, {lower: true}) + '-' + (Math.random() * Math.pow(36, 6) |
0).toString(36)
  }
}
```

```
import {ApiBearerAuth, ApiImplicitFile, ApiOperation, ApiResponse, ApiUseTags}
from "@nestjs/swagger";
import {Controller, HttpException, HttpStatus, Post, UploadedFile, UseInterceptors}
from "@nestjs/common";
import {FileInterceptor} from "@nestjs/platform-express";
import {PhotosService} from "./photos.service";

@ApiBearerAuth()
@ApiUseTags('photos')
@Controller('photos')
```

```
export class PhotosController {
   constructor(private readonly photosService: PhotosService) {}

   @ApiOperation({ title: 'Upload photo' })
   @ApiResponse({ status: 201, description: 'Uploaded'})
   @ApiResponse({ status: 403, description: 'Forbidden' })
   @ApiImplicitFile({ name: 'image' })
   @Post('upload')
   @UseInterceptors(FileInterceptor('file'))
   async uploadPhoto(@UploadedFile() file: any) {
      try{
         const fileFormat =file.originalname.split('.').pop();
         await this.photosService.upload(file.buffer, fileFormat);
         return await new HttpException('Uploaded', HttpStatus.CREATED);
      }catch (e) {
         return e
      }
   }
}

import {Column, Entity, PrimaryGeneratedColumn} from "typeorm";

export enum PhotoStatuses {
   Uncompressed,
   Compressed,
}

@Entity('photo')
export class PhotoEntity {

   @PrimaryGeneratedColumn()
   private id: number;

   @Column()
   pathUncompressed:string;

   @Column()
   private pathCompressed: string;

   @Column({default: PhotoStatuses.Uncompressed})
```

```
    status: PhotoStatuses;
}

import {MiddlewareConsumer, Module, NestModule, RequestMethod} from
"@nestjs/common";
import {TypeOrmModule} from "@nestjs/typeorm";
import {PhotoEntity} from "./photos.entity";
import {PhotosService} from "./photos.service";
import {PhotosController} from "./photos.controller";
import {AuthMiddleware} from "../user/auth.middleware";
import {UserModule} from "../user/user.module";

@Module({
    imports: [TypeOrmModule.forFeature([PhotoEntity]), UserModule],
    providers: [PhotosService],
    controllers: [
        PhotosController
    ],
    exports: []
})
export class PhotosModule implements NestModule {
    public configure(consumer: MiddlewareConsumer) {
        consumer
            .apply(AuthMiddleware)
            .forRoutes({path: 'photos', method: RequestMethod.ALL});
    }
}


import {HttpException, HttpStatus, Injectable} from "@nestjs/common";
import {InjectRepository} from "@nestjs/typeorm";
import {Repository} from "typeorm";
import {PhotoEntity, PhotoStatuses} from "./photos.entity";
import * as fs from 'fs';
import * as sharp from 'sharp';

@Injectable()
export class PhotosService {
    constructor(
        @InjectRepository(PhotoEntity)
```

```
      private readonly photosRepository: Repository<PhotoEntity>,
   ) {}

   async upload(file: ArrayBuffer, format: string) {

if(format==='jpg'||format==='png'||format==='jpeg'||format==='webp'||format==='tiff'||fo
rmat==='gif'||format==='svg'){
         const fileName = new Date().getTime() + 'uncompressed.'+format;
         const ws = await fs.createWriteStream(fileName);
         console.log(file);
         fs.writeFileSync(fileName, file);
         await this.photosRepository.insert({ pathUncompressed: fileName, status:
PhotoStatuses.Uncompressed });

      }
      else {
         throw new HttpException('Image format is not supportered',
HttpStatus.BAD_REQUEST);
      }
   }

   async startWorker() {
      setInterval(async () => {
         console.log('Starting compression');
         const uncompressedPhoto = await this.photosRepository.findOne({ status:
PhotoStatuses.Uncompressed });
         if (!uncompressedPhoto) {
            console.log('No uncompressed photos, skipping');
            return;
         }
         console.log(uncompressedPhoto);
         const fileFormat =uncompressedPhoto.pathUncompressed.split('.').pop();
         const compressedPath = new Date().getTime() + '.'+fileFormat;
         await
sharp(uncompressedPhoto.pathUncompressed).withMetadata().toFile(compressedPath);
         uncompressedPhoto.status = PhotoStatuses.Compressed;
         uncompressedPhoto.pathCompressed = compressedPath;
         await this.photosRepository.save(uncompressedPhoto);
         fs.unlinkSync(uncompressedPhoto.pathUncompressed);
      }, 15000);
```

```
    }
}

import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";
import {IsEmail, Validate} from "class-validator";
import * as crypto from 'crypto';
// import { CustomEmail } from '../user/CustomEmail';

@Entity('follows')
export class FollowsEntity {

  @PrimaryGeneratedColumn()
  private id: number;

  @Column()
  followerId: number;

  @Column()
  followingId: number;

}


import { Get, Post, Delete, Param, Controller } from '@nestjs/common';
import { Request } from 'express';
import { ProfileService } from './profile.service';
import { ProfileRO } from './profile.interface';
import { User } from '../user/user.decorator';

import {
  ApiUseTags,
  ApiBearerAuth,
} from '@nestjs/swagger';

@ApiBearerAuth()
@ApiUseTags('profiles')
@Controller('profiles')
export class ProfileController {

  constructor(private readonly profileService: ProfileService) {}
```

```
  @Get(':username')
  async getProfile(@User('id') userId: number, @Param('username') username: string):
Promise<ProfileRO> {
    return await this.profileService.findProfile(userId, username);
  }

  @Post(':username/follow')
  async follow(@User('email') email: string, @Param('username') username: string):
Promise<ProfileRO> {
    return await this.profileService.follow(email, username);
  }

  @Delete(':username/follow')
  async unFollow(@User('id') userId: number,  @Param('username') username: string):
Promise<ProfileRO> {
    return await this.profileService.unFollow(userId, username);
  }

}

export interface ProfileData {
  username: string;
  bio: string;
  image?: string;
  following?: boolean;
}

export interface ProfileRO {
  profile: ProfileData;
}

import {MiddlewareConsumer, Module, NestModule, RequestMethod} from
'@nestjs/common';
import { ProfileController } from './profile.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ProfileService } from './profile.service';
import { UserModule } from '../user/user.module';
import {UserEntity} from "../user/user.entity";
import {FollowsEntity} from "./follows.entity";
```

```
import {AuthMiddleware} from "../user/auth.middleware";

@Module({
  imports: [TypeOrmModule.forFeature([UserEntity, FollowsEntity]), UserModule],
  providers: [ProfileService],
  controllers: [
    ProfileController
  ],
  exports: []
})
export class ProfileModule implements NestModule {
  public configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(AuthMiddleware)
      .forRoutes({path: 'profiles/:username/follow', method: RequestMethod.ALL});
  }
}


import { HttpStatus, Injectable} from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { UserEntity } from '../user/user.entity';
import { DeepPartial } from 'typeorm/common/DeepPartial';
import { ProfileRO, ProfileData } from './profile.interface';
import {FollowsEntity} from "./follows.entity";
import {HttpException} from "@nestjs/common/exceptions/http.exception";

@Injectable()
export class ProfileService {
  constructor(
    @InjectRepository(UserEntity)
    private readonly userRepository: Repository<UserEntity>,
    @InjectRepository(FollowsEntity)
    private readonly followsRepository: Repository<FollowsEntity>
  ) {}

  async findAll(): Promise<UserEntity[]> {
    return await this.userRepository.find();
  }
```

```
  async findOne(options?: DeepPartial<UserEntity>): Promise<ProfileRO> {
    const user = await this.userRepository.findOne(options);
    delete user.id;
    if (user) delete user.password;
    return {profile: user};
  }

  async findProfile(id: number, followingUsername: string): Promise<ProfileRO> {
    const _profile = await this.userRepository.findOne( {username:
followingUsername});

    if(!_profile) return;

    let profile: ProfileData = {
      username: _profile.username,
      bio: _profile.bio,
      image: _profile.image,
    };

    const follows = await this.followsRepository.findOne( {followerId: id, followingId:
_profile.id});

    if (id) {
      profile.following = !!follows;
    }

    return {profile};
  }

  async follow(followerEmail: string, username: string): Promise<ProfileRO> {
    if (!followerEmail || !username) {
      throw new HttpException('Follower email and username not provided.',
HttpStatus.BAD_REQUEST);
    }

    const followingUser = await this.userRepository.findOne({username});
    const followerUser = await this.userRepository.findOne({email: followerEmail});

    if (followingUser.email === followerEmail) {
```

```
      throw new HttpException('FollowerEmail and FollowingId cannot be equal.',
HttpStatus.BAD_REQUEST);
    }

    const _follows = await this.followsRepository.findOne( {followerId: followerUser.id,
followingId: followingUser.id});

    if (!_follows) {
      const follows = new FollowsEntity();
      follows.followerId = followerUser.id;
      follows.followingId = followingUser.id;
      await this.followsRepository.save(follows);
    }

    let profile: ProfileData = {
      username: followingUser.username,
      bio: followingUser.bio,
      image: followingUser.image,
      following: true
    };

    return {profile};
  }

  async unFollow(followerId: number, username: string): Promise<ProfileRO> {
    if (!followerId || !username) {
      throw new HttpException('FollowerId and username not provided.',
HttpStatus.BAD_REQUEST);
    }

    const followingUser = await this.userRepository.findOne({username});

    if (followingUser.id === followerId) {
      throw new HttpException('FollowerId and FollowingId cannot be equal.',
HttpStatus.BAD_REQUEST);
    }
    const followingId = followingUser.id;
    await this.followsRepository.delete({followerId, followingId});

    let profile: ProfileData = {
```

```
      username: followingUser.username,
      bio: followingUser.bio,
      image: followingUser.image,
      following: false
    };

    return {profile};
  }

}


import {PipeTransform, ArgumentMetadata, BadRequestException, HttpStatus,
Injectable} from '@nestjs/common';
import { validate } from 'class-validator';
import { plainToClass } from 'class-transformer';
import {HttpException} from "@nestjs/common/exceptions/http.exception";

@Injectable()
export class ValidationPipe implements PipeTransform<any> {
  async transform(value, metadata: ArgumentMetadata) {

    if (!value) {
      throw new BadRequestException('No data submitted');
    }

    const { metatype } = metadata;
    if (!metatype || !this.toValidate(metatype)) {
      return value;
    }
    const object = plainToClass(metatype, value);
    const errors = await validate(object);
    if (errors.length > 0) {
      throw new HttpException({message: 'Input data validation failed', errors:
this.buildError(errors)}, HttpStatus.BAD_REQUEST);
    }
    return value;
  }

  private buildError(errors) {
```

```
    const result = {};
    errors.forEach(el => {
      let prop = el.property;
      Object.entries(el.constraints).forEach(constraint => {
        result[prop + constraint[0]] = `${constraint[1]}`;
      });
    });
    return result;
  }

  private toValidate(metatype): boolean {
    const types = [String, Boolean, Number, Array, Object];
    return !types.find((type) => metatype === type);
  }
}


import { SECRET } from '../config';
import * as jwt from 'jsonwebtoken';

export class BaseController {

  constructor() {}

  protected getUserIdFromToken(authorization) {
    if (!authorization) return null;

    const token = authorization.split(' ')[1];
    const decoded: any = jwt.verify(token, SECRET);
    return decoded.id;
  }
}

import { Test } from '@nestjs/testing';
import { TagController } from './tag.controller';
import { TagService } from './tag.service';
import {TypeOrmModule} from "@nestjs/typeorm";
import {TagEntity} from "./tag.entity";

describe('TagController', () => {
```

```
  let tagController: TagController;
  let tagService: TagService;

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      imports: [TypeOrmModule.forRoot(), TypeOrmModule.forFeature([TagEntity])],
      controllers: [TagController],
      providers: [TagService],
    }).compile();

    tagService = module.get<TagService>(TagService);
    tagController = module.get<TagController>(TagController);
  });

  describe('findAll', () => {
    it('should return an array of tags', async () => {
      const tags : TagEntity[] = [];
      const createTag = (id, name) => {
        const tag = new TagEntity();
        tag.id = id;
        tag.tag = name;
        return tag;
      }
      tags.push(createTag(1, 'angularjs'));
      tags.push(createTag(2, 'reactjs'));

      jest.spyOn(tagService, 'findAll').mockImplementation(() => Promise.resolve(tags));

      const findAllResult = await tagController.findAll();
      expect(findAllResult).toBe(tags);
    });
  });
});

import {Get, Controller, Post} from '@nestjs/common';

import { TagEntity } from './tag.entity';
import { TagService } from './tag.service';

import {
```

```
  ApiUseTags,
  ApiBearerAuth,
} from '@nestjs/swagger';

@ApiBearerAuth()
@ApiUseTags('tags')
@Controller('tags')
export class TagController {

  constructor(private readonly tagService: TagService) {}

  @Get()
  async findAll(): Promise<TagEntity[]> {
    return await this.tagService.findAll();
  }


}

import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert} from "typeorm";
import {IsEmail, Validate} from "class-validator";
import * as crypto from 'crypto';
// import { CustomEmail } from './CustomEmail';

@Entity('tag')
export class TagEntity {

  @PrimaryGeneratedColumn()
  private id: number;

  @Column()
  private tag: string;

}


import {MiddlewareConsumer, Module, NestModule, RequestMethod} from
'@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UserModule } from '../user/user.module';
```

```
import { TagService } from './tag.service';
import { TagEntity } from './tag.entity';
import { TagController } from './tag.controller';

@Module({
  imports: [TypeOrmModule.forFeature([TagEntity]), UserModule],
  providers: [TagService],
  controllers: [
    TagController
  ],
  exports: []
})
export class TagModule implements NestModule {
  public configure(consumer: MiddlewareConsumer) {
  }
}
```

```
import { Injectable} from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { TagEntity } from './tag.entity';

@Injectable()
export class TagService {
  constructor(
    @InjectRepository(TagEntity)
    private readonly tagRepository: Repository<TagEntity>
  ) {}

  async findAll(): Promise<TagEntity[]> {
    return await this.tagRepository.find();
  }

}
```

```
import { HttpException } from '@nestjs/common/exceptions/http.exception';
import { NestMiddleware, HttpStatus, Injectable } from '@nestjs/common';
import { ExtractJwt, Strategy } from 'passport-jwt';
```

```
import { Request, Response, NextFunction } from 'express';
import * as jwt from 'jsonwebtoken';
import { SECRET } from '../config';
import { UserService } from './user.service';

@Injectable()
export class AuthMiddleware implements NestMiddleware {
  constructor(private readonly userService: UserService) {}

  async use(req: Request, res: Response, next: NextFunction) {
    const authHeaders = req.headers.authorization;
    if (authHeaders && (authHeaders as string).split(' ')[1]) {
      const token = (authHeaders as string).split(' ')[1];
      const decoded: any = jwt.verify(token, SECRET);
      const user = await this.userService.findById(decoded.id);

      if (!user) {
        throw new HttpException('User not found.', HttpStatus.UNAUTHORIZED);
      }

      req.user = user.user;
      next();

    } else {
      throw new HttpException('Not authorized.', HttpStatus.UNAUTHORIZED);
    }
  }
}


import { Get, Post, Body, Put, Delete, Param, Controller, UsePipes } from
'@nestjs/common';
import { Request } from 'express';
import { UserService } from './user.service';
import { UserEntity } from './user.entity';
import { UserRO } from './user.interface';
import { CreateUserDto, UpdateUserDto, LoginUserDto } from './dto';
import { HttpException } from '@nestjs/common/exceptions/http.exception';
import { User } from './user.decorator';
import { ValidationPipe } from '../shared/pipes/validation.pipe';
```

```
import {
  ApiUseTags,
  ApiBearerAuth,
  ApiOperation
} from '@nestjs/swagger';

@ApiBearerAuth()
@ApiUseTags('user')
@Controller()
export class UserController {

  constructor(private readonly userService: UserService) {}

  @Get('user')
  async findMe(@User('email') email: string): Promise<UserRO> {
    return await this.userService.findByEmail(email);
  }

  @Put('user')
  async update(@User('id') userId: number, @Body('user') userData: UpdateUserDto) {
    return await this.userService.update(userId, userData);
  }

  @UsePipes(new ValidationPipe())
  @Post('users')
  async create(@Body() userData: CreateUserDto) {
    return this.userService.create(userData);
  }

  @Delete('users/:slug')
  async delete(@Param() params) {
    return await this.userService.delete(params.slug);
  }

  @UsePipes(new ValidationPipe())
  @Post('users/login')
  async login(@Body() loginUserDto: LoginUserDto): Promise<UserRO> {
    const _user = await this.userService.findOne(loginUserDto);
```

```
    const errors = {User: 'not found'};
    if (!_user) throw new HttpException({errors}, 401);

    const token = await this.userService.generateJWT(_user);
    const {email, username, bio, image} = _user;
    const user = {email, token, username, bio, image};
    return {user}
  }
}


import { createParamDecorator } from '@nestjs/common';
import { SECRET } from '../config';
import * as jwt from 'jsonwebtoken';

export const User = createParamDecorator((data, req) => {

  // if route is protected, there is a user set in auth.middleware
  if (!!req.user) {
    return !!data ? req.user[data] : req.user;
  };

  // in case a route is not protected, we still want to get the optional auth user from jwt
  const token = req.headers.authorization ? (req.headers.authorization as string).split(' ') :
null;
  if (token && token[1]) {
    const decoded: any = jwt.verify(token[1], SECRET);
    return !!data ? decoded[data] : decoded.user;
  }

});


import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert, JoinTable,
ManyToMany, OneToMany} from "typeorm";
import { IsEmail, Validate } from 'class-validator';
import * as crypto from 'crypto';
import { ArticleEntity } from '../article/article.entity';
import {ExerciseEntity} from "../exercise/exercise.entity";
```

```
@Entity('user')
export class UserEntity {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  username: string;

  @Column()
  @IsEmail()
  email: string;

  @Column({default: ''})
  private Name: string;

  @Column({default: ''})
  bio: string;

  @Column({default: ''})
  image: string;

  @Column()
  password: string;

  @ManyToMany(type => ArticleEntity)
  @JoinTable()
  private favorites: ArticleEntity[];

  @OneToMany(type => ArticleEntity, article => article.author)
  articles: ArticleEntity[];

  @OneToMany(type => ExerciseEntity, exercise => exercise.author)
  private exercises: ExerciseEntity[];
}


export interface UserData {
  username: string;
  email: string;
```

```
  token: string;
  bio: string;
  image?: string;
}

export interface UserRO {
  user: UserData;
}

import {MiddlewareConsumer, Module, NestModule, RequestMethod} from
'@nestjs/common';
import { UserController } from './user.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UserEntity } from './user.entity';
import { UserService } from './user.service';
import { AuthMiddleware } from './auth.middleware';

@Module({
  imports: [TypeOrmModule.forFeature([UserEntity])],
  providers: [UserService],
  controllers: [
    UserController
  ],
  exports: [UserService]
})
export class UserModule implements NestModule {
  public configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(AuthMiddleware)
      .forRoutes({path: 'user', method: RequestMethod.GET}, {path: 'user', method:
RequestMethod.PUT});
  }
}


import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository, getRepository, DeleteResult } from 'typeorm';
import { UserEntity } from './user.entity';
import { CreateUserDto, LoginUserDto, UpdateUserDto } from './dto';
```

```
const jwt = require('jsonwebtoken');
import * as bcrypt from 'bcrypt';
const saltRounds = 10;
const myPlaintextPassword = 's0/\/\P4$$w0rD';
const someOtherPlaintextPassword = 'not_bacon';
import { SECRET } from '../config';
import { UserRO } from './user.interface';
import { validate } from 'class-validator';
import { HttpException } from '@nestjs/common/exceptions/http.exception';
import { HttpStatus, UnauthorizedException } from '@nestjs/common';
import * as crypto from 'crypto';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(UserEntity)
    private readonly userRepository: Repository<UserEntity>
  ) { }

  async findAll(): Promise<UserEntity[]> {
    return await this.userRepository.find();
  }

  async findOne(loginUserDto: LoginUserDto): Promise<UserEntity> {

    const findOneOptions = {
      email: loginUserDto.email
    };

    const user = await this.userRepository.findOne(findOneOptions);
    if (!user) {
      const errors = { user: 'User not found!' };
      throw new HttpException({ message: 'User not found!', errors },
HttpStatus.BAD_REQUEST);

    }
    const passwordValid = await bcrypt.compare(loginUserDto.password,
user.password);
    console.log(user.password);
    if (!passwordValid) {
```

```
      const errors = { password: 'Invalid password!' };
      throw new HttpException({ message: 'Invalid Password!', errors },
HttpStatus.BAD_REQUEST);
    }

    return this.generateJWT(user)

  }

  async create(dto: CreateUserDto): Promise<UserRO> {

    // check uniqueness of username/email
    const { username, email, password } = dto;
    const qb = await getRepository(UserEntity)
      .createQueryBuilder('user')
      .where('user.username = :username', { username })
      .orWhere('user.email = :email', { email });

    const user = await qb.getOne();

    if (user) {
      const errors = { username: 'Username and email must be unique.' };
      throw new HttpException({ message: 'Input data validation failed', errors },
HttpStatus.BAD_REQUEST);

    }

    // create new user
    let newUser = new UserEntity();
    newUser.username = username;
    newUser.email = email;
    newUser.password = await bcrypt.hash(password, saltRounds);
    newUser.articles = [];

    const errors = await validate(newUser);
    if (errors.length > 0) {
      const _errors = { username: 'Userinput is not valid.' };
      throw new HttpException({ message: 'Input data validation failed', _errors },
HttpStatus.BAD_REQUEST);
```

```
  } else {
    const savedUser = await this.userRepository.save(newUser);
    return this.buildUserRO(savedUser);
  }

}

async update(id: number, dto: UpdateUserDto): Promise<UserEntity> {
  let toUpdate = await this.userRepository.findOne(id);
  delete toUpdate.password;
  delete toUpdate.favorites;

  let updated = Object.assign(toUpdate, dto);
  return await this.userRepository.save(updated);
}

async delete(email: string): Promise<DeleteResult> {
  return await this.userRepository.delete({ email: email });
}

async findById(id: number): Promise<UserRO> {
  const user = await this.userRepository.findOne(id);

  if (!user) {
    const errors = { User: ' not found' };
    throw new HttpException({ errors }, 401);
  };

  return this.buildUserRO(user);
}

async findByEmail(email: string): Promise<UserRO> {
  const user = await this.userRepository.findOne({ email: email });
  return this.buildUserRO(user);
}

public generateJWT(user) {
  let today = new Date();
  let exp = new Date(today);
  exp.setDate(today.getDate() + 60);
```

```
    return jwt.sign({
      id: user.id,
      username: user.username,
      email: user.email,
      exp: exp.getTime() / 1000,
    }, SECRET);
  };

  private buildUserRO(user: UserEntity) {
    const userRO = {
      id: user.id,
      username: user.username,
      email: user.email,
      bio: user.bio,
      token: this.generateJWT(user),
      image: user.image
    };

    return { user: userRO };
  }
}
```

# Додаток Б

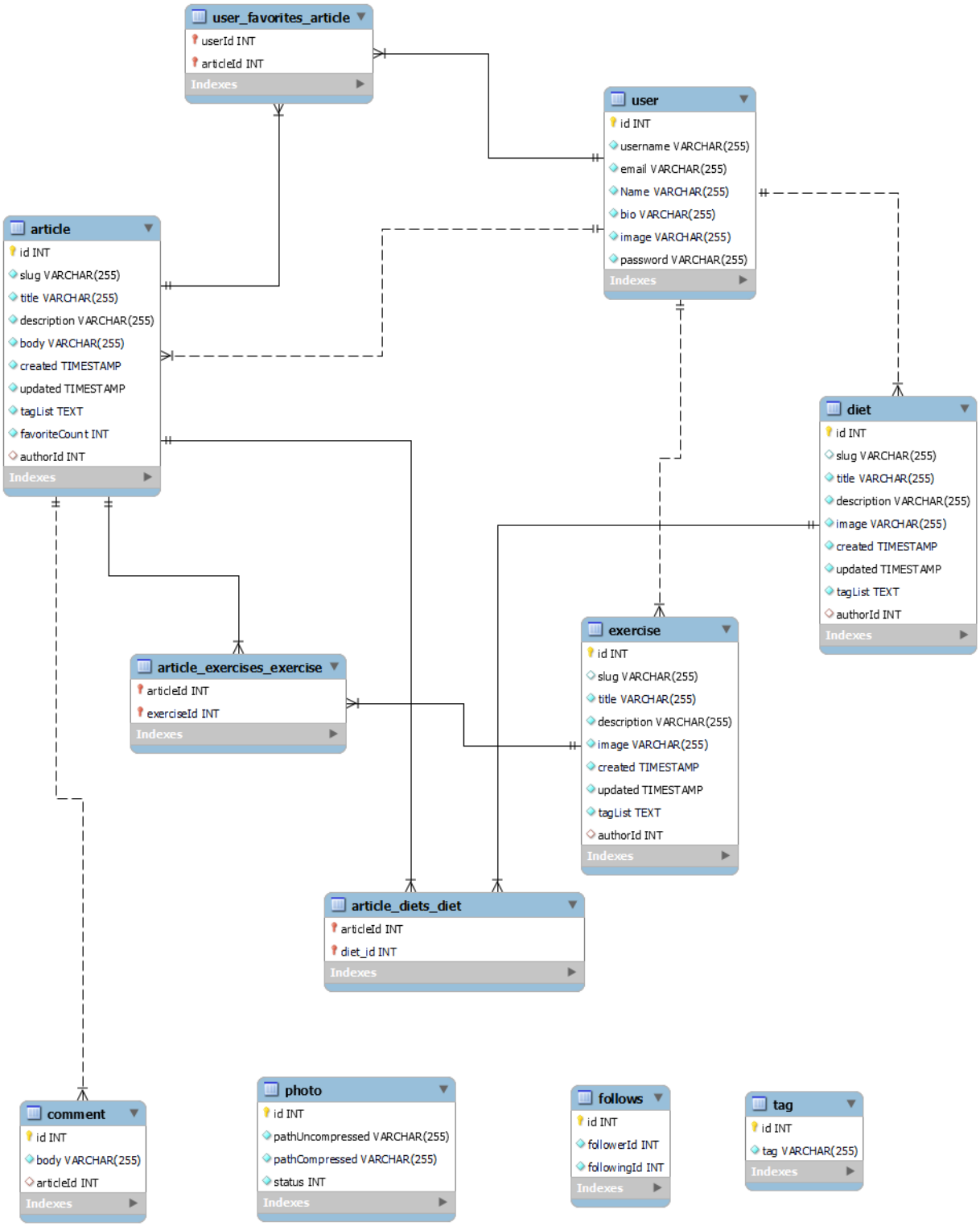



Рисунок Б.1 – Діаграма бази даних

# Додаток В

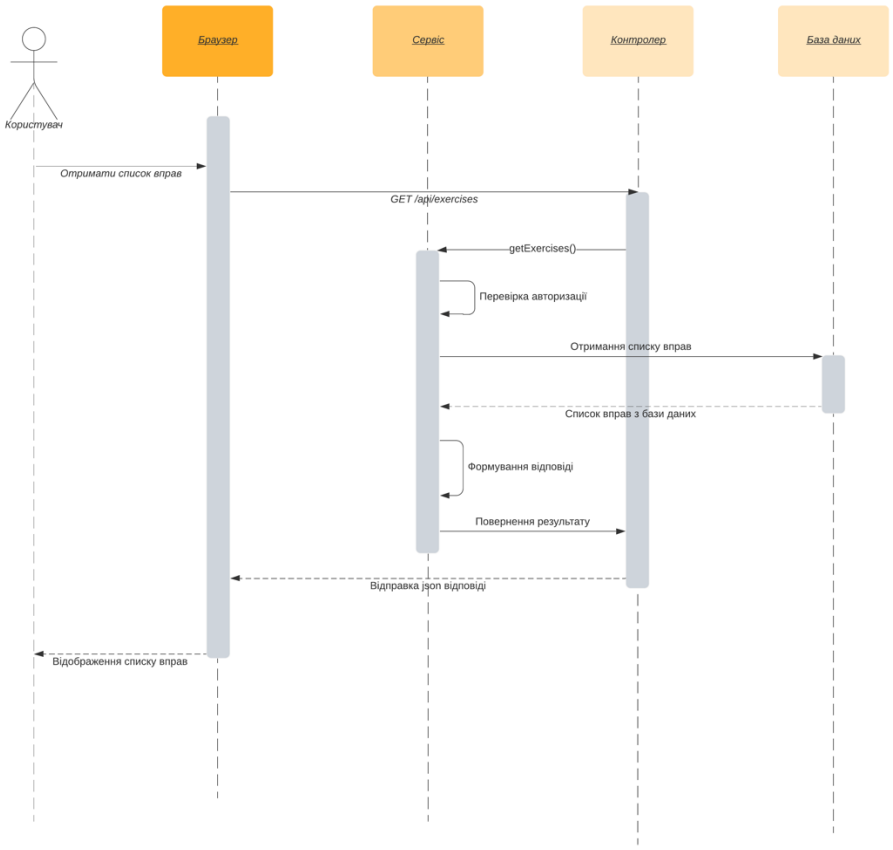



Рисунок В.1 – Діаграма послідовностей

# Додаток Е

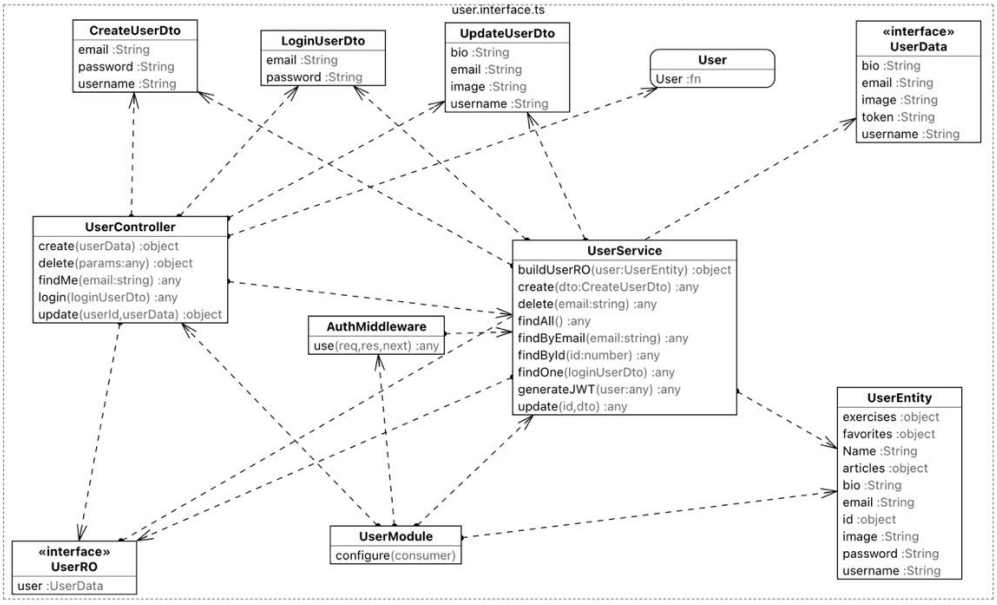



Рисунок Е.1 –

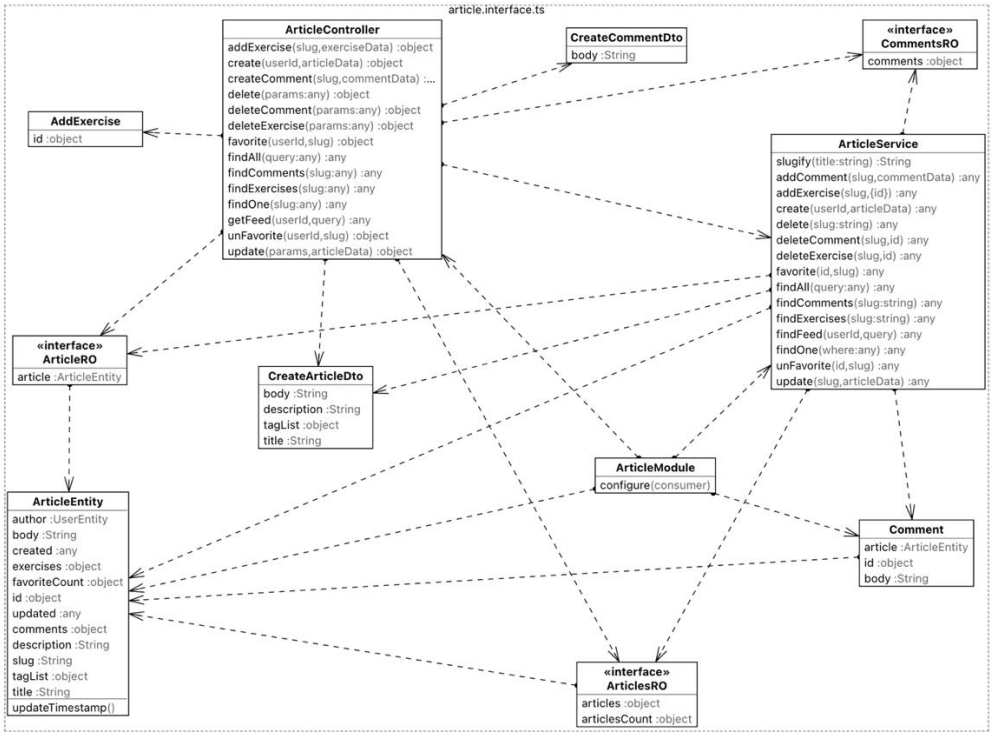



Рисунок Е.2 –

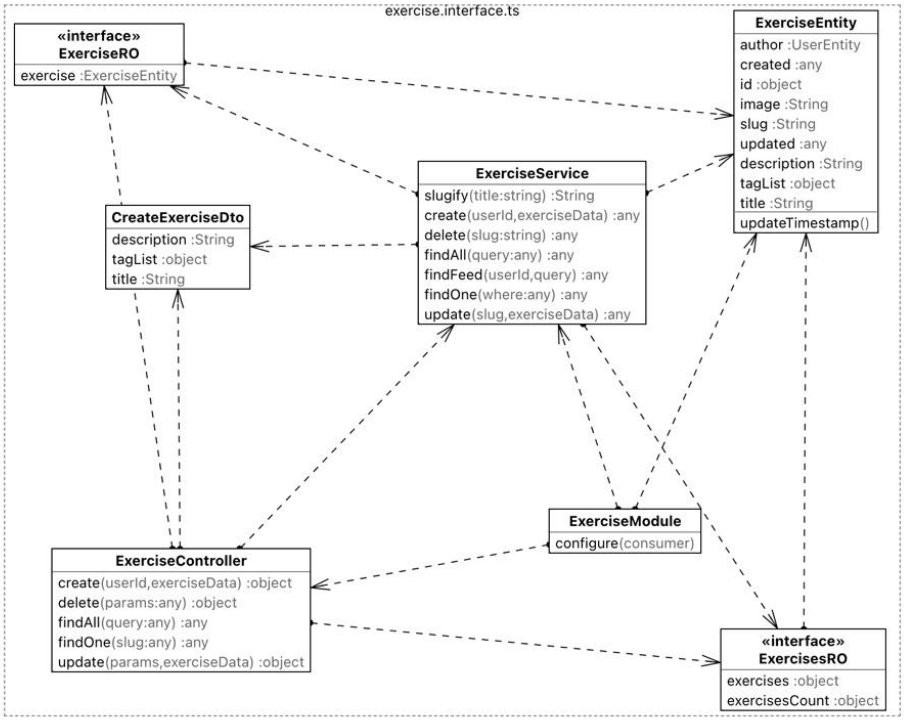



Рисунок Е.3 –

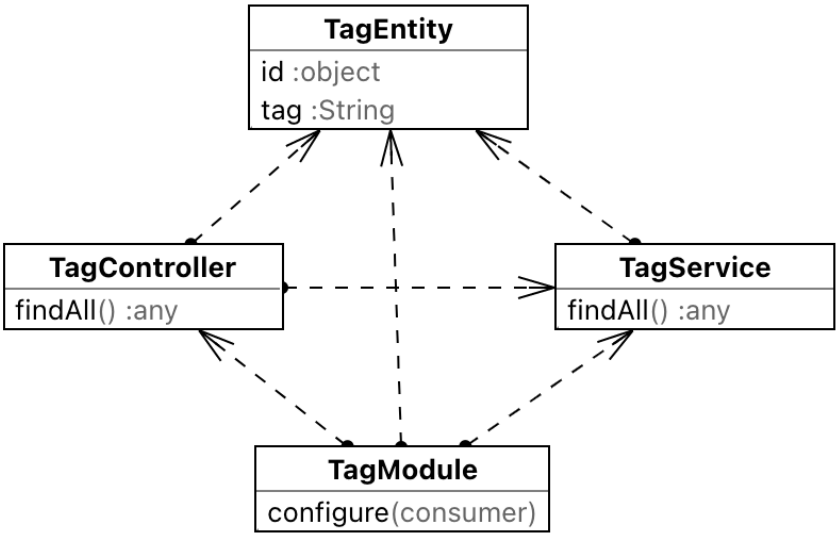



Рисунок Е.4 –

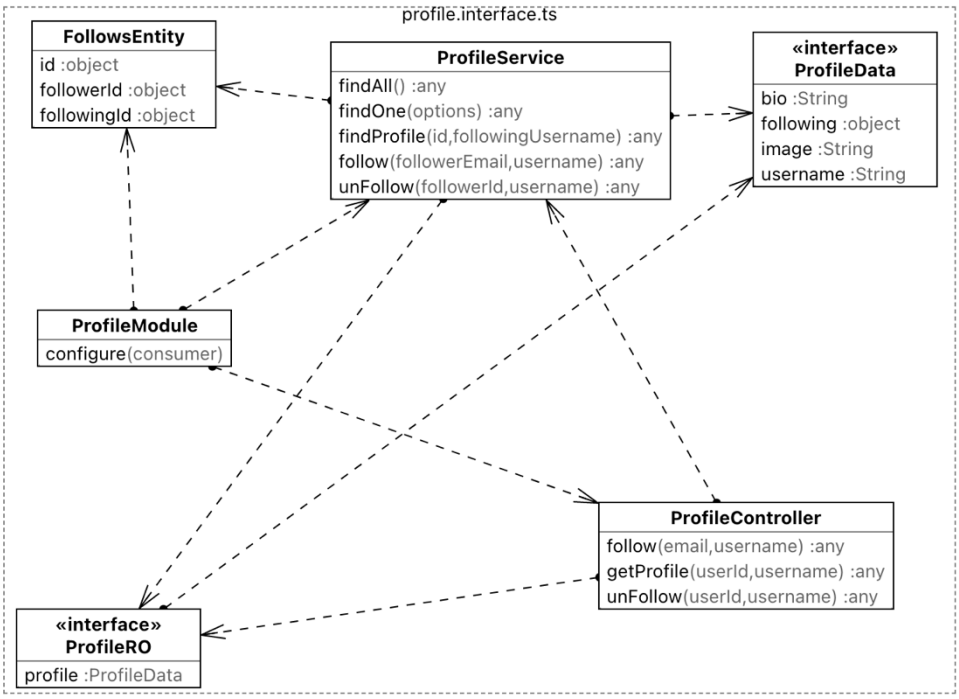



Рисунок Е.5 –

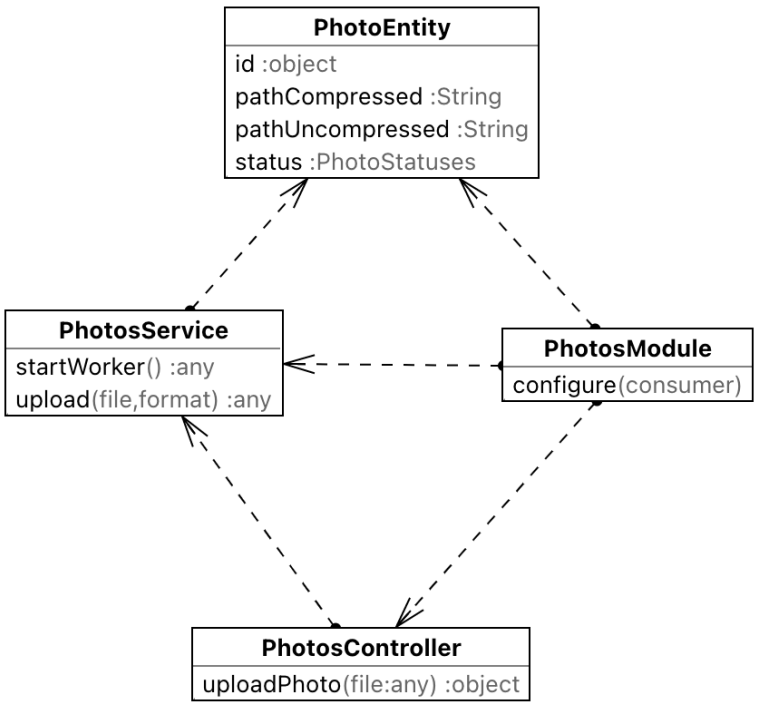



Рисунок Е.6 –